The Pragmatic Programmers

TURING 图灵程序设计丛书

Beyond Legacy Code

Nine Practices to Extend the Life (and Value) of Your Software

修改软件的艺术

构建易维护代码的9条最佳实践

[美] David Scott Bernstein 著　李满庆 译

作者30年软件开发经验总结

揭示高质量软件的秘密，阐述真正的敏捷开发之道



中国工信出版集团　人民邮电出版社
POSTS & TELECOM PRESS

数字版权声明

作 者

David Scott Bernstein

敏捷教练，曾为IBM、微软、Yahoo等企业提供敏捷实践指导。他的公司To Be Agile（tobeagile.com）指导团队进行测试先行、结对编程以及重构等极限编程实践。

译 者

李满庆

Web开发者，现就职于百度，关注Web技术、编程语言、敏捷开发。希望能通过推广先进的开发实践来改善现在开发流程中普遍存在的一些问题，提高开发者的生产力。

TURING 图灵程序设计丛书

Beyond Legacy Code
Nine Practices to Extend the Life (and Value) of Your Software

修改软件的艺术

构建易维护代码的9条最佳实践

[美] David Scott Bernstein 著　李满庆 译

人民邮电出版社
北　京

图书在版编目（C I P）数据

修改软件的艺术 : 构建易维护代码的9条最佳实践 / (美) 戴维•斯科特•伯恩斯坦著 ; 李满庆译. -- 北京 : 人民邮电出版社, 2017.10
（图灵程序设计丛书）
ISBN 978-7-115-46776-8

Ⅰ. ①修… Ⅱ. ①戴… ②李… Ⅲ. ①软件开发 Ⅳ. ①TP311.52

中国版本图书馆CIP数据核字(2017)第217905号

内 容 提 要

本书会帮你降低构建与维护软件的成本。如果你是软件开发者，将学到一套实践方法以构建易修改的代码，因为在应用当中代码经常需要修改。对于和软件开发者合作的管理者来说，本书会向你展示为何引入这 9 个基本的实践方法，会使你的团队更加有效地交付软件而不至于让软件演变成遗留代码。

本书适合软件开发人员和 IT 经理阅读。

◆ 著　　　　[美] David Scott Bernstein
译　　　　李满庆
责任编辑　朱　巍
执行编辑　张海艳
责任印制　彭志环

◆ 人民邮电出版社出版发行　　北京市丰台区成寿寺路11号
邮编　100164　　电子邮件　315@ptpress.com.cn
网址　http://www.ptpress.com.cn
北京　　　　　　　印刷

◆ 开本：800×1000　1/16
印张：12
字数：284千字　　　　2017年 10 月第 1 版
印数：1－3 500册　　　2017年 10 月北京第 1 次印刷
著作权合同登记号　图字：01-2016-9522号

定价：55.00元

读者服务热线：(010)51095186转600　印装质量热线：(010)81055316

反盗版热线：(010)81055315

广告经营许可证：京东工商广登字 20170147 号

版权声明

本书赞誉

本书从全新的视角展现了现代软件开发流程。工程师们会在其中找到解决日常问题的方案，而非工程师们可以对软件开发中所面对的挑战和困难有所认识。

——Stas Zvinyatskovsky，埃森哲公司资深首席软件架构师

David帮我们认清了我们是如何陷入此番境地的。他给出了行之有效的理论和工具，也提出了一些值得深刻思考的问题。对于关心软件开发的人来说，本书是一份厚礼。要善用它。

——Ron Jeffries，RonJeffries.com

如果你想要优化软件交付流程，但是感觉到裹足不前、无能为力，那么这本书正适合你。对于开始频繁迭代交付以及尝试采用敏捷却未见显著效果的人来说，这是本好书。

——Gojko Adzic，Neuri Consulting LLP公司合伙人

本书讨论的内容可以帮我让客户更加满意，也能让他们在需求出现变化时一直开心。

——David Weiser，Moz软件工程师

对于所有的开发者和管理者来说，这都是一本好书，而且适用于各类公司的各类代码。

——Troy Magennis，Focused Objective CEO

David的解释清晰明了，我甚至希望开发团队的管理者以及开发定制软件的公司领导们也能看看这本书。理解这些实践，将能构建出更经济、更易维护和扩展的软件。

——Jim Fiolek，黑骑士金融服务公司软件架构师

书中的各种观点令人欣慰。如果可以让人们都遵从这些原则，我们的生活以及软件开发会变得更加轻松惬意。

——Nick Capito，Unboxed Technology公司软件开发总监

我们努力让每一行代码成为真实产品的一部分。要找出让我们误入歧途的原因。要找到合适的方法，让你的团队在现在以及今后能更有效地开发出客户真正想要的产品。

——Michael Hunter，极客，骇客，首席工程师，架构师

序

遗产

遗产是已经死亡的事物存留下的依旧影响着世界的那部分。

能留下遗产的生命是优秀的，但是软件并非如此。我们用温和的词语“遗留”来形容那些已经失去活力但是依旧运行的代码，让那些过去的决策持续影响着那些深陷其中的人。

软件和硬件要区分对待。我们称硬件为“硬”是因为它是固定的，没有工具是无法调整的。软件的“软”是指它由思想而生，通过代码来表达，加载到硬件中然后行使一些职责。

讽刺的是，代码在编写完成脱离开发者之后变得比硬件还难修改。

开发者通过编程的逻辑表达想法和需求，赋予软件生命。就像是无中生有一样，直到我们意识到所有的这些小心论证都是为了让这个新生命成为我们所期望的那样。

敏捷

面对威胁与机遇的时候，反应迅速的组织被称为“敏捷”组织。一个敏捷的组织会吸取经验教训，不会被那些短期内不会修改的软件束缚手脚，无法行动。

一些思想者，包括我在内，选择“敏捷”这个词来形容我们都希望软件能拥有即刻适应需求变化的能力。这股敏捷的势力在新开发的软件中很强烈，而且不以任何形式消失，因为软件如此重要，它应该在这整个软件生命周期中持久旺盛。

这些思想者向那些“算命”的管理实践提出异议，它们要求软件开发者在一个合作的流程中对未来进行预估，然后根据那些预估的准确与否评判开发者。

这些思想者向那些“提前大规模设计”的开发实践提出异议，它们需要预料到所有可能的情况才能保证后面维护不至于出现“大问题”。

然后这些思想者将他们的建议用“软件开发宣言”的形式发布给开发者和管理者，这宣言在今天比在十多年前发布的时候吸引了更多的注意力。

宣言并不完美。它在两个方面有所欠缺。它的短小让读者以为它仅仅是些一般性观点而不是具体编写软件的特定建议。

同时，宣言标题中的“开发”也让读者认为仅仅是新项目才需要采用，到上线时就不需要了。

奏效

受雇的开发者必须意识到，仅仅言听计从是不够的，他们有义务让交付的软件持续产生价值。

软件的复杂度可能因为不当的处理而加剧，成为程序员的负担，难以逃避。大部分的敏捷方法，尤其是本书，描述了在接受这种现实的组织中工作的方式。

这本书温习了那些实践，它们在最近二十年中被一遍又一遍证实了既有效但又难以实施。

“名义上的敏捷”形容那些采用了在无数的书中描述的实践，却收效甚微的组织。本书将这些实践进行解剖，分析其背后的原因。只有牢牢地把握住敏捷的逻辑，才能判断“名义上的敏捷”的问题所在。

计算成本的不断下降，让真正的高质量软件的价值凸显了出来。哪里商业更迅速，哪里就有财富。在边界逐渐瓦解之时，任何生意都需要优秀的软件才能生存。

奇怪的是，激增的廉价计算机让编程更难了。多数的敏捷开发者，所谓的“多语言程序员”，知道他们的技艺在敏捷组织之外很难得到欣赏。

David Bernstein解释了为什么敏捷方法会起作用。他深入挖掘了自身的经验，用自己的故事展示了这些实践的价值。

《修改软件的艺术》明确了成功使用这些实践并从中获取最大价值的前提。

Ward Cunningham

美国俄勒冈州波特兰市

引　言

本书会帮你降低构建与维护软件的成本。

如果你是软件开发者，将学到一套实践方法以构建易修改的代码，因为代码在应用当中经常需要修改。对于和软件开发者合作的管理者来说，本书会向你展示为何引入这9个基本的实践方法，会使你的团队更加有效地交付软件，而不至于让软件演变成遗留代码。为此，你需要的不仅仅是一份技术性的任务清单，还需要对为什么有这些实践方法以及如何实施这些实践方法有着深刻的理解。

每天，我们都会因为遗留代码而损失时间、金钱和机遇。

不同的人对“遗留代码”有着不同的定义，但是简而言之，遗留代码就是指因为种种原因，格外难以修正、改进以及使用的代码。

这样的代码有很多。实际上我所见过的所有生产环境下的软件几乎都是遗留代码。

软件产业通常轻视可维护性，所以到最后，企业花在维护代码上的成本比一开始编写代码的成本还高。正如我们将在第2章看到的，软件开发的低效仅在美国每年就造成百亿美元级的开销，这可不仅仅是某份报告上的抽象数字。我们每天都受到遗留代码的影响。软件很昂贵，容易出错，而且难以改进。

业内外人士已经开始拉帮结派为某些项目管理方法论争论不休（其中不乏杰出的点子），但是为了更好地做出持久性的变革，我们首先要对软件开发的基础目标达成共识。

本书不仅仅是关于如何构建更好的软件，更是关于如何构建更好的软件产业。书中囊括了我身为专业开发者三十年所学的精华。我从业的头二十年都是在传统的瀑布式开发下度过的，系统分别按照设计、构建、测试的阶段开发。问题是，规划软件开发的方式充满了不可预见的问题，这迫使我们必须对质量和预算做出严重的妥协。

但是在最近的十年间，对我以及其他我认识的开始尝试**极限编程**（Extreme Programming，XP）的软件开发者来说，事情发生了改变。采用这种敏捷开发方法论之后，我们放弃试图在一开始把所有事情想明白，而是循序渐进地来做，每次只设计、构建、测试一小部分软件。

极限编程中的一些实践方法，诸如测试驱动开发和重构，在构建和扩展软件过程中降低风险

和成本方面给我上了重要的一课。这些实践方法的应用呈现给我各种各样的解决软件问题的方法。是否能利用这些实践方法，揭示出构建高质量、高可维护的软件的方法呢？

我的回答是响亮的“能”！

在程序员生涯初期，我被指派去从标准普尔的feed中整理股票数据，并将数据发送到客户的私有数据库中。在此之前，这一过程都是手动完成的，容易出错而且平均每天需要花费十四小时完成。我需要将这一过程自动化，但是对于如何找到最佳的解决方案，一开始我却摸不着头脑。

几周之后的某一天，在已经写了四十多页代码之后，我突然灵光一闪，有了重组数据处理方式的想法。在几个小时之内，我就完成了这个项目并且将代码削减到只剩下五页。那天早上开工时预计要花几个月完成的工作，结果在当天下班前就完成了。从那时起，我有过多次灵光一闪，甄别出藏在问题之下的规律，而这些规律让我能够迅速地构建出高可维护的解决方案。

这仅仅是一个例子，说明同一问题的不同解决方式之间有着巨大的效率差别。我从其他开发者那里听到过许多类似的故事。也许你也有灵光一闪之后，问题化繁为简的故事。

以我的专业经验，效率极高的开发者和一般开发者之间的差别可以非常大。我花了职业生涯中的大部分时间，研究那些罕见的、比一般软件开发者效率高数倍的个例。我了解到，那些人并非生来如此。他们仅仅是形成了一些和我们不同的特质，也许是遵循一些不同寻常的实践方式。所有的这些都是可以后天习得的技能。

作为一个新生的产业，我们依然在摸索着，并且学着去抓住重点。构建软件和构建实体物品不同。也许，软件产业面对的一些挑战乃是植根于对软件开发本质的理解误区。为了理解软件开发，使其可以预估，软件开发被拿来和制造业以及土木工程相比较。尽管软件工程和其他工程领域有着相似点，但是有些基础性的差异对那些不是日常编写软件的人来说不那么显而易见。

软件工程和其他形式的工程不一样，这一点并不为奇。医学和法学并不一样，木工和烘焙并不一样。软件开发与且只与一个东西一样，那就是软件开发。我们需要一些实践方法，使我们能够更高效、更可校验、更容易做出更改。如果我们能做到这些，就可以削减构建软件的短期成本，并且完全消除维护它的那些可怕的长期成本。

为此，我给出来自于极限编程、Scrum和精益等敏捷方法论的9种实践方法。当这9种实践方法不仅被应用而且被彻底理解的时候，它们可以帮助我们避免编写的代码在将来变成遗留代码。

尽管还有很多近乎无法修复或者已经濒临废弃的代码，我们依然可以利用相同的实践方法，慢慢地在已经堆积成山的遗留代码之中找到出路。

这9种实践方法将帮助开发团队构建更好的软件，并且帮助整个产业避免浪费金钱、时间和资源。

我见证过客户如何受益于这些实践方法。那些客户曾构建了一些有史以来最大、最复杂的软件。我知道使用这些实践方法可以产生非凡成效，但是仅仅“使用”它们并不能确保有成效。为

了正确使用它们，我们必须理解这些实践方法背后的道理。

这是一个令人着迷的时代，而我们身在其中。当我们作为先行者在未知的领域里探险时，会有灯光为我们照亮方向。本书中的9种实践方法照亮了我的软件开发之路。我希望它们也成为你的指路明灯。

如何使用本书

本书探讨软件开发，但并不是非得成为软件开发者才能理解本书。

软件的编写过程对于大多数人来说可能是个陌生的概念，但是它却影响着我们所有人。它已经成为了一个如此复杂的活动，以至于开发者经常发现自己在试图给客户甚至经理解释各种概念，可能没有一点参考。本书有助于架起沟通的桥梁，用通俗的语言解释技术概念，帮我们在究竟什么是优秀的软件开发这个问题上达成共识。

尽管让不同类型的读者针对技术实践方法达成共识并非易事，但本书的设计目标是为了帮助5种不同的人群对软件开发拥有同样的理解：

- 软件开发者
- 软件开发和 IT 经理人
- 软件购买者
- 各个行业的产品经理和项目经理
- 其他所有对这个重要技术感兴趣的人

我试图让软件开发对于所有人来说都容易理解，通过讲述故事并利用大量的轶事、类比和隐喻将技术概念形象化。软件开发难以一概而论，所以从我所说的话中找出反例是很容易的，但是通常都会有更深层次的见解尚待发现。

为了让非开发者也能够阅读本书并关注这些实践方法的重要性，本书并没有按照教程方式编写。关于撰写用户故事以及重构等各个方面，已经有许多优秀图书了（见参考文献）。尽管本书也提供了很多实际的建议，但是它最特别也是最有价值的地方是探讨了**为什么**这些技术实践方法会有用。这种方式能让那些非开发者（如管理人员和其他利益相关者）明白开发者在构建软件期间面临的问题和挑战。

第一部分：遗留代码危机

在第一部分，我们将直面软件产业发展的若干重大问题，并且发现由于糟糕的软件开发流程导致每年数十亿美元流失。

大部分维系我们生存的软件易出错、脆弱并且近乎无法扩展，也就是我们所谓的“遗留代

码”。这一部分将探讨我们是如何变成现在这样的，这样的情况又意味着什么。探讨不仅针对软件产业，而且针对所有与其相关的人和产业。

如果你已经对软件产业耳熟能详，甚至已经深感疲惫，将会对这些内容“倍感亲切”。你将会更加深刻地理解为什么在软件构建过程中，事情总无法按照计划执行，以及为什么需要更好的方法。

即使对于业内人士来说，第一部分也可以帮助他们从适当的角度来看待这些巨大挑战。管理人员和开发者都可能会从这些问题中发现全新的视点，而这些问题对我们这个产业来说是每天都在发生的。正如一位经理对我说的那样：“它使我的炮火更加精准。”本书可以帮你说服他人：在解决问题之前，至少我们必须识别出问题所在。

如果你是软件行业的局外人，我敢保证第一部分会让你称奇，甚至震惊。

第二部分：延续软件生命（和价值）的 9 种实践方法

在第二部分中，由于之前已经说明了问题所在，本书剩下的四分之三将从阴暗压抑中走出，介绍一套实践方法，提供真实、可操作的解决方案，首先是一些对管理者特别有用的实践方法。

在第5章和第6章，我将提出一些自己实践后总结的建议，不仅针对如何更好地开始实施复杂软件的开发流程，还针对如何一路把控流程直到完成。这两个实践方法对于软件行业以外的人会非常有用，同时我也提供了适用于任何项目管理环境的建议。实施这些实践方法后，你将能够：

- 更加高效地执行任务
- 节约短期和长期成本
- 构建更高质量的软件
- 增加客户的满意度，带来回头客

接下来是我在职业生涯中发现的非常有效的7个技术性实践方法。这些方法更适合软件开发者。

我见过这些实践方法的成功与失败。有些软件开发团队运用了最好的实践方法，但是技术很差，所以无法获取所期望的价值。成功运用这些实践方法的团队和那些失败的团队之间的差异是，失败的团队没有弄明白这些实践方法为什么如此重要。而这正是本书所强调的。

即便这些是技术性的实践方法，我也想敦促管理者们（任何行业的管理者）以开放的心态了解这些基本的概念。了解你管理的开发者所面对的挑战，跟他们分享这些实际的、可以迅速上手的实践方法，帮助在破碎不堪的流程中苦苦挣扎的团队提高效率与效益。

当你读到这些实践方法的描述时，我希望你在思考如何在项目中实施之前，先想想这些实践方法为什么有其价值，这样会帮助你更加有效地运用这些实践方法。

虽然我建议阅读（并应用）这9个实践方法，但是可以按照任意的顺序去执行。我当然意识到本书的每个读者都有具体的问题和需求，所以我将第二部分组织为9个独立的实践方法。专注于那些对你最有帮助和能快速生效的方法，但是不要止步于此。

我并不想独树一帜，而是想抛砖引玉

如何阅读本书，从中有什么样的收获，全在于你。我努力避免玩弄“敏捷”“Scrum”“极限编程”这样的词汇。我希望本书改变人们对于软件开发这个新兴产业的看法，使其进入主流的视野。我希望在软件开发社区里展开讨论，这个社区常常拉帮结派对细节争论不休但缺乏大局观，我们不应该做井底之蛙，而应该基于一个共同目的分享想法，这个目的就是：尽一切可能构建最优秀的软件。

线上资源

本书的代码示例可以在Pragmatic Programmers网站上的本书主页（http://pragprog.com/book/dblegacy）找到。你也可以在论坛中提问并得到回答，还会发现一个可以提交错误报告的勘误表，等等。

致　　谢

首先要感谢我的妻子 Staci Bernstein给了我编写本书所需的空间以及一切支持，让我可以完成夙愿，还有我们的迷你杜宾犬Nicki，它在我写作、编辑的时候时刻陪伴着我。

其次我要感谢 Ward Cunningham 为本书编写了序，感谢他给予我的所有鼓励。

写一本书是一项重大的任务，而本书依仗许多人的支持，对此我深表感激。感谢我的伙伴、同事，以及客户在我写作此书过程中给予的探讨和支持。

我还要对本书的审阅者深表感谢，他们将自己的时间和经验无私地投入本书，提出宝贵意见，才让本书更加优秀，尤其感谢 Scott Bain，Heidi Helfand，Ron Jeffries，Jim Fiolek，Stas Svinyatskovsky，Ed Kraay，James Couball，Pat Reed，Stephen Vance，Rebecca Wirfs-Brock，Jeff Brice，Jerry Everand，Greg Smith，Ian Gilman，Llewellyn Falco，Fred Daoud，Michael Hunter，Woody Zuill，GojkoAdzic，Troy Magennis，Kevin Gisi，David Weiser，Nick Capito，Sam Who，Michael Hunger，Ken Pugh，Max Guernsey，以及 Chris Sterling。

最后，我要感谢在过去三十年中参加过我课程的数千名软件开发者。我非常感激从你们那里所学到的东西！

目　　录

第一部分

遗留代码危机

只有我一个人这样吗？其他人是否也注意到——或者在乎——有那么多的软件并未按照期望的那样工作，或者没有正常工作多长时间，而且近乎无法修补。

软件远远达不到 100% 的成功。但是软件行业到底情况如何？有多少软件项目是成功的？我们对于成功和失败的定义又是什么？如何度量它们？

我向很多人问过这些问题，软件行业内外的人都有。许多软件行业外的人觉得这样问很奇怪：“你说有些项目竟然不成功？”但是软件行业内的人却倾向于问：“你的意思是有的项目竟然成功了？”

第 1 章

有些事情不对劲

有些事情不对劲。

组织内部毫无信任可言。繁重的软件开发流程处处碍事，让他们再也无法生产代码。整个公司陷入死亡的漩涡，总价值七亿五千万美元的生意危在旦夕。

你也许是这个团队核心开发人员中的一位。这些核心开发人员十分杰出，但是同时也会有二等的初级开发人员，以及一些编外的或者第二梯队的团队，他们允许混入一些“码农气息”的短见，只关注眼前一个功能，而不是这个功能如何与整个系统整合，而且意识不到，他们在做的这些事情中有些将会在短期内引发大问题，并且在以后导致更加严重的后果。

即使由聪明的、经验丰富的专业开发者带领开发工作，创造出的软件依然不符合标准并且难以使用。整个开发团队不清楚技术实践背后的原因。他们在这里偷工减料，那里钻个空子，建立一个个“小团队”甚至各自为战，各自使用不同的标准，而对整个系统只有片面的认识。这让代码集成变成了没人愿意看到的噩梦般的体验。

或者你是管理者中的一员，负责让这个软件开发完成、交付、盈利而非亏损。这些管理者聪明而且经验丰富，但到头来也和开发者一样，身心俱疲，甚至失去信心。公司管理者看到最后期限被推迟，眼看着不稳定的预发布版本被推上线，而又不知道说些什么能帮开发者们把事情做好。所以管理层的应对是加入更多的流程，愈发地破坏团队的信任，进而让最后期限一再推迟。

在各种组织中，开发人员、质量保证人员、实施人员之间总是会衍生出敌意，这家苦苦挣扎的公司就是这样。开发者和管理者都不明白为什么他们要反过头来重新审视他们的工作，事实是他们的燃尽图让他们不得不这样。

我是他们雇佣的第三个顾问了，而且是第一个不把他们的问题归结为“人为因素”的人。我看到的是一个**遗留代码**的问题。他们的软件脆弱且难以使用。公司在过去的十年中飞速发展，他们的代码却因此受到了连累。

公司曾经尝试“敏捷”过几年，但是即使他们的许多团队施行了一些敏捷实践，他们已有的代码依旧碍事，使得预估不准确进而拖慢进度。他们知道必须解决他们已有的遗留代码，以及那些一路养成的让他们陷入这般境地的坏习惯。

我们关注本书中的这些实际的工程实践，以及**它们的原理**。当我说这些开发者和管理者是聪明的经验丰富的专业人士的时候，我不是在开玩笑。他们认真倾听，将自己投入到一个需要自我改变的合作过程中去。他们对修改软件和流程的问题下了必要的功夫。

如果你是这些开发者中的一员，你将发现你不再会在凌晨三点钟由于系统停机被叫起来。你开始变得迅速，从你添加的测试得到正确反馈，让你明白你的代码正如你所预期那样执行。与团队的其他成员一起，将20%的时间花在清理已有的代码上，你们会见证这些努力在将来的一年内得到的丰厚回报。

如果你是管理团队的一分子，你会看到团队成员合作更加有效率。你将不会再担惊受怕由于一些“关键资源”（主要开发者）决定换个工作，剩下的人无法维护那个人的代码，进而使公司遭受打击。你会看到团队中开发者的态度发生了变化，开始处理遗留代码问题而非视而不见。

虽然花了几个月，但是随着代码的改进，团队的速度也变快了。团队的规划变得更稳妥，他们开始连续地按期完成而不偷工减料。

他们开始打破心理上和空间上的围墙了。部门之间开始沟通。他们通力协作，开始重构质量保证和需求控制的方式。测试人员和开发人员坐在一起想办法自动化测试他们的待发布版。最终，从耗时两周的大量手动测试过程变成一个完整的自动化测试过程，大部分情况下只花费不到两分钟。这样做，每年为公司节约了大量的金钱，也为组织结构改革提供了基础。

简而言之一句话：人们开始上心了。

这是一个真实的故事。我看着它一次又一次上演。如果你是软件开发者，或者你管理软件开发者，害怕正朝这个死亡漩涡一头栽下，那么本书会让你了解这个公司是如何扭转局面的。如果你正挣扎求生，要知道，你不是一个人在战斗。

软件开发和维护的方式中有什么东西不对劲。但是并不是非得如此。

1.1 什么是遗留代码

软件开发是世上独一无二的工作。当我们理解了它的本质以及它需要持续更新的特性后，就可以找到许多方式来增强所编写的代码的健壮性，进而降低维护和扩展的成本。

Michael Feathers在他的《修改代码的艺术》[Fea04]一书中，提出了当我们听到“遗留代码”的时候会想到什么：

> 如果你也和我一样，那么大抵会联想到错综复杂的、难以理清的结构，需要改变然而实际上又根本无法理解的代码；你会联想到那些不眠之夜，试图添加一个本该很容易就添加上去的特性；你会联想到自己是如何的垂头丧气，以及你的团队中的每个人对一个似乎没人管的代码库是如何打心底里感到厌烦的，这种代码简直让你生不如死。你内

心深处甚至对于想一想怎样才能改善这种代码都感到痛苦。这种事情似乎太不值得我们付出努力了。

我们知道软件是如何演变成遗留代码的。和其他事物一样，软件也有一定的生命周期。程序被创造、使用、修补，最终淘汰。软件像生物一样，如果它赖以生存的操作系统被淘汰，也会死亡。正如医生一样，软件开发者能做的最多是推迟大限的到来。如果病患术后的生活质量比之前得到了提升，则治疗是成功的。但我们都清楚，我们所有人都有大限将至的一天，软件也一样。

在一个程序的生命中，代码被修修改改，使得原本的设计变得脆弱，所以软件变得越来越难用。由于许多现如今编写的软件通常都是难以变更的，最后我们往往是替换代码而不是修复代码。

一种全新的被称为“软件考古学”的领域因这种情况而兴起，这一名称在2002年被《程序员修炼之道》的作者Dave Thomas和Andy Hunt提出[1]。当我面对一个多年前构建的、没有文档、变量命名糟糕的软件时，有时候会觉得，考古学家在从一块陶片中窥视一个失落文明的秘密的时候一定也是这种感觉。简直是无以为继。

当我们深入观察软件构建过程中的缺点时，可以发现这些缺点是如何导致遗留代码的产生和繁衍的。如果可以意识到预估一件我们从未做过的事情的时间、成本和进度是多么具有挑战性的事情，并意识到软件工程和其他工程学巨大的差异，我们才能开始了解遗留代码是从哪里来的以及该如何应对。

Michael Feathers进一步把遗留代码定义为任何没有测试的代码。这是因为他和我一样，十分重视优秀的自动化单元测试，这些测试可以使代码更健壮并保证其表现得和预期一样。

良好的单元测试的前提是你有优秀的、可测试的代码，但对于遗留代码来说经常并非如此，所以你必须清理代码，使其处于更好的状态。这通常说起来比做起来容易。让不可测试代码变为可测试可能需要重新架构整个系统，即使有些技术手段作为辅助，也需要大量的工作。

对于遗留代码，没有简单的答案，没有快速的修复方式。在第2章中，我们会看到到底这个问题有多么普遍，又是对软件产业造成了多少损失。一个如此巨大的问题让我们需要退后一步，从整个不同的角度去审视它。如果过去实施的方法并未奏效，那么我们也许需要寻找其他的解决方案。

1.2　顺流直下

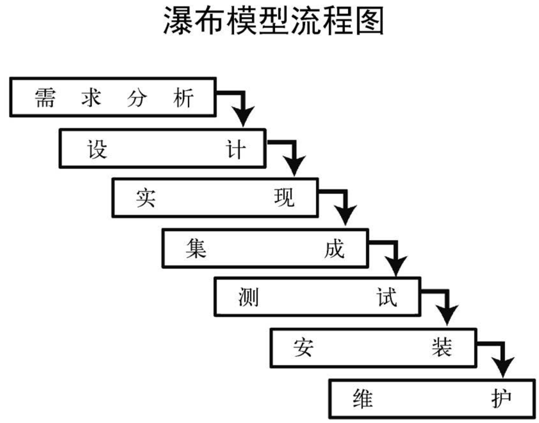
瀑布模型是从制造业和建筑行业借鉴而来，最早于1970年由Winston Royce提出[2]。它是一系列用于软件构建的阶段——但是他紧跟着又在下一页说这样的流程不会起作用。显而易见，从来

① Hunt, Andy, and Thomas, Dave. “Software Archaeology.” *Software Construction/IEEE Software* March/April 2002. http://media.pragprog.com/articles/mar_02_archeology.pdf

② http://agileconsortium.pbworks.com/w/page/52184647/Royce%20Defining%20Waterfall

没人读到过那里。

一度作为软件构建的主流方法，瀑布模型的概念很简单，提供了7个独立的步骤，按照如下顺序进行。



需求分析

从对应领域的专家和潜在用户那里收集信息，从而建立一个需求文档。需求文档是一系列要求，指导我们在当前发布版本中应该实现哪些功能。功能就是软件的种种**职责**。

设计

接着是要根据写好的需求文档去设计软件。这些设计的形式通常是设计图例和其他**表达**设计思想的产出物。这并非代码，相当于另一个文档：如何构建软件的图例和描述。和蓝图不一样，蓝图是对建筑的每个细节的事无巨细的表达，但是软件的架构远谈不上精确或者全面。

实现

在设计之后是实现阶段，这个阶段代码被编写出来以满足设计。编码就是单纯地完成设计产出物中描述的设计。

集成

在所有代码编写完毕之后开始集成阶段。在集成阶段中，所有团队成员编写的代码放到一起。这通常是第一次所有代码被编译到一个计算机程序中。

测试

一旦软件集成完毕则测试阶段开始，验证软件表现是否如预期的那样。这个阶段包括对软件

执行一系列的测试用于证明软件正常工作。

安装

在安装阶段，软件发布给用户。此阶段可能包括将载有程序的CD邮寄给用户或者在线上提供软件下载。

维护

最后，就是对软件进行持续维护：修复问题，添加新功能，提供更新。

瀑布模型在桥梁建造或者制造小物品时意义重大，因为将需求整合投入生产的方式更加高效。但软件开发并不是一个制造业的流程。软件开发者不是通过预先生产好的部件组装程序。当然，有些软件部件可以提前构建，但是大部分我们所需的组件要自己构建或者修改，甚至研发。直到真正需要使用之前，我们很少知道要构建、修改或研发什么样的组件，更不用说如何加固我们的架构了。

1.3　孤注一掷

我们在瀑布模型开发的项目中持续累积风险的方式，和在拉斯维加斯赌博的方式有着惊人的相似。

在传统的瀑布开发中，为了让**一个**部分正常工作，**所有**其他部分必须都能正常工作。程序员并不知道他们的代码和系统其他部分配合得如何，直到集成阶段——发布前最后的阶段之一——所有的独立代码才组合到一起。

直到最后阶段才做集成，基本上是在玩轮盘赌，而且连中十次才算赢。

整个过程中一个错误，甚至一行代码的问题，就会造成整个程序没法编译成可执行代码，或者编译成功但在执行的时候崩溃。这就是我们所谓的“bug”。

许多bug都是**只**在集成阶段才会出现。我们在项目的最后阶段才集成代码，这给开发过程带来极大的不确定性。由于已经进入集成阶段，任何必要的修改和奇怪的bug都会成为严重且代价高昂的问题，需要付出大量的努力（以及附带的成本）去修复。我难以想象，还会有其他编写软件的方式比这样风险更高、更容易出错。

这就是为什么在软件开发中最后一刻的修改会有如此昂贵的代价。它需要大量的人力去重新测试和集成代码。有可能一个局部的小改动会影响程序的其他部分，所以通常情况下都会谨慎地重新测试整个系统，无论系统中哪个部分作了修改。如果整个过程是手动进行的，成本会极高，而且会阻碍最后一刻的一切修改，这些修改可能来自于重要的信息或者是开发后期才产生的更好的主意。

1.4　为什么瀑布模型不管用

当软件以一个长的发布周期构建的时候，开发者可能在他编写代码几个月后才能看到它执行。开发者可能会建立一个测试场景来让他们的代码可以被调试器（一种帮你检查其他软件中bug的软件）管辖，然后一条语句一条语句地执行，但是这和代码本来应该在的整个系统的上下文环境中执行完全不一样。

这是将不同功能打包成成品的构建方式的主要缺点之一。先部分后整体非常符合直觉，这是我们建造东西的方式。如果你在盖房子，你会希望打地基需要用的所有东西都在手边，而不是停下来等着混凝土或者其他东西被送来。你也会希望所有的木材到位，不用花钱请木匠干坐着等顶梁或者其他什么被送来。

这是现实事物和虚拟事物之间主要不同之一。事实证明，先部分后整体的方式**在虚拟世界并不奏效**。

并不仅仅是因为这种方式效率低下——我们会在考察不同发布周期优劣的时候讨论这种低效——而是它让我们建造出来的东西**难以改变**。这个问题看上去非常不明显，却至关重要。

仅当增量构建时，你可以在其中增加些伸缩缝以便后期扩展。你在优化构建持续发布软件过程中从没这么想过。在瀑布模型中，这从来不是重点也不是问题。没人会在盖房子的时候想着以后加上额外的房间。蓝图怎么画的你就怎么盖。现实中又有多少人会想在房子里增加额外的房间呢？相比之下，向软件中加入新功能又是多么得频繁。

1.4.1　食谱与配方

食谱和配方有所不同。举例来说，你可以根据**食谱**制作意式番茄罗勒酱，它尝起来和其他厨师按照同一食谱制作的酱一样，但是仅仅是在严格遵循食谱的前提下。如果一个厨师加了点胡椒，另一个厨师多用了点罗勒少用了些牛至，虽然做出来的依然是意式番茄罗勒酱，但味道不同。另一方面来说，如果两个面包师烤面包，其中一个改变了水、面粉和酵母的比例，面包就会发不起来——面包也就完蛋了。烘焙面包需要的是**配方**。

我们必须停止将软件开发视为配方，应严格按照细节执行，将其看作食谱，让不同的主厨可根据独有的理解依不同情况做出调整。

除了其中一些最乏味的任务以外，编程并不是一种“按图索骥”的活动。许多任务需要开拓新领地，进入没有前人涉足的地方。软件开发有许多**禁忌**，在一种场景下“正确”的方法在另一场景却会失败。

这种前途未知的情况迫使传统的瀑布流程为了预估和简化诸多的未知而变得越来越复杂。但是，使某样东西更复杂并不会自动让其变得更好，而且可能很容易就变得碍手碍脚。

1.4.2　开发和测试分离

“我的工作是尽我所能、尽快地完成安排给我的所有功能，而质量保证团队是我的守护天使，所以我可以草率些，大胆地去尝试，因为我的工作是做出粗糙的草案，然后交给这个‘共同作者’去完善它。”

作为软件开发人员，有多少次你这么想过或说过？或者你是管理人员，听人这么说过？但是质量保证团队真的不是软件开发人员的“共同作者”。质量保证人员所能说的就是：“回去再试试。哎，顺便说一句，你没什么时间了。”

将自己置于如此境地，我们该如何取胜？但是我们却习以为常，这是一个严重的问题，因为这会鼓励开发者不认真关注他们正在做的事情。

问题并非局限于瀑布模型开发。多数的软件开发项目依然有独立的质量保证工作，对待发布的版本进行耗时数天甚至数周的手动测试。

刺激与反馈要尽可能紧密结合以便改变我们的习惯。当开发者直到几个月后才看到行为的结果的时候，它已经变得不完全合适了。就像我们的人生座右铭：

> 找出错误并非我的工作；创造错误才是。

正如瀑布模型一样，诸如“六西格玛”和“全面质量管理”等其他流行的项目管理方法学，在商品制造业上对持续性和质量都有着非凡成效，但应用在软件方面却一败涂地。讽刺的是，我们花了大量时间在注重方法论本身的检查和协调，却忽视了提高产品本身的质量。

传统的软件开发实践，关注着对未来的预计以及可预估的工作流，结果开发者和管理者都规避了过程中的繁杂特性，而仅仅关注于完成任务。

1.5　当“流程”变成“体力劳动”

当我还是IBM的程序员的时候，我们有个规矩，每行代码都需要包含一句来自开发者的注释。这无意中鼓励了开发者对数据和函数使用糟糕的命名规范，因为可以使用注释来“解释”代码。人们开始阅读注释而非代码。在时间紧迫的时候，开发者常常更新代码而忽略了注释，使得代码和注释不同步。过期的注释比没有注释更加糟糕。这会使得代码成了谎言，而我们不希望代码中出现谎言。

假设有如下代码：

```
x++; /* 此处 x 增一 */
```

我们并不需要“/* 此处 x 增一 */”这样的注释。我们可以假设读者理解编程语言的基础。冗余的注释在好的情况下是噪音，在糟糕的情况下则变为谎言，无论是否故意为之。

看到大量的“流水账注释”——那些并非描述代码**缘由**而是描述代码**行为**的注释——的时候，我会意识到写出这样注释的开发者格外担心阅读代码的人是否理解其代码的所作所为。代码应该是自解释的，应该通过优秀的命名和通行的用法使得软件易于理解。

冗余的注释成了编写低质量代码的借口。用块注释为一段代码提供大段的描述——而不是将该行为放入独立的私有方法中使其能通过见名知意的方法名来调用——会使代码难以阅读而且导致方法职责过多。

有充分的理由相信，IBM在这个注释规则背后的初衷是好的。事实上，我认为整个软件开发的“地狱之路”都是由良好的初衷所铺筑的，除非我们理解了软件开发的真正本质，我们实施的“解决方案”往往注定适得其反。

比如，我知道有一家公司在编写一行代码之前需要写十二个主要文档并且要求所有的部门主管签字，因为之前的失败导致了管理层和开发者之间明显缺乏信任。讽刺的是，他们大部分问题的根源正是繁复的流程，而非开发人员的不负责任。管理层以增加更多的流程作为应对，反而让事情进一步错乱。

应该假设在任何项目的开始时期，相关人员都是想尽力交付高质量产品的。但是不知为何，我们从准备充分且乐于奉献演变成了疏远而对立——准备钻钻空子。

人们在对事情的结果没有影响力或者仅仅**觉得**自己没有影响力的时候，就会感到被疏远了。这样的事可能每时每刻在我们身边发生着。反之，让人们参与到某件事情中的最有效方法可以总结为一个词：尊重。

1.6 坚如磐石的管理

现代管理技术发展乃是工业革命的结果。那些早期的管理者们拿着秒表在工厂里面溜达，为做着不同任务的工人们计算时间，然后给出一个简单命令：**快点！**

但是在软件开发中，没有那些可以用一组标准评判完成质量的重复性任务让我们可以干得“快点”。当然，我们的手指在键盘上忙碌，但实际上是我们的脑子在工作，形象化，模型化，然后用代码使这些模型具体化。每个任务都是与众不同的，寻求解决方案需要不停学习新东西。

管理者希望保证他们的开发人员在做着正确的事情，但是更深层次的问题是：

究竟什么是“正确的事情”？

我们构建软件的目标是什么？遵循什么样的原则？许多其他领域的专业人士都能轻易回答这个问题，但是我未见几个软件开发者能给出答案。

为了回答“什么是‘正确的事情’”，管理者倾向于通过提高生产率（早期的产品线管理者用秒表），或者强制执行更复杂的计划，即使这样的做法往往适得其反，最后消磨人们的斗志。我

们加入的流程越多情况越糟糕，因为**流程无法支配创造力**。我们必须认清软件开发本质上是一个创造过程。

1.7　此处有龙

我们可以利用地图出行，计划旅途，可以清晰地标出A点和B点以及其间的最短路径。即使拉丁词语Hic sunt draconnes（意即“此处有龙”）没有如大部分人认为的那样出现在古代地图上，它也成为了对于有待探索的未知领域的象征。那些位于地图边缘的地方十分恐怖，充满了怪兽……而软件开发者深陷其中。

众所周知，依赖数字、图表和最后期限能让管理层——其实是几乎所有的人——感到安心。但是想在任何专业领域做出伟大事迹，都意味着要挑战未知，而未知是难以量化的。

既然软件开发常常在充满“龙”的土地上进行，我们为何要度量软件开发呢？又是如何度量？度量什么？

进行度量是为了找出待改进的部分，增加可预测性，减少风险等。除此之外，是为了在前往未知领域的征途中感到安心。度量是为了能合理地猜测项目所需的时间和成本。

但是无论多么合理，终究还是猜测。

最终，我们进行度量是让我们自以为一切尽在掌握之中。

然而这些都是错觉。

猜测和错觉可以给我们的前行提供信心，但同样会给我们虚假的信息，让我们误入歧途，而后为之付出代价。

不难想象，我们管理的项目挑战性越大，在过程中做出的量化就越不精确，至少有时如此。当我们构建虚拟产品时，必须习惯对过程有着更抽象的理解，在大多数人并不理解虚拟产品开发的情况下保证进度。

传统上来说，软件开发管理者需要创建用于系统设计、编码和测试的时间表。这种分段方法让每个人都能了解项目进展情况。不幸的是，这种方式时常演变成谎言。一个九个月的项目进行了八个月时，我们发现进度比预期晚了六个月。这一切是如何发生的呢？

其实我们一直是晚了六个月，只不过我们到了第八个月才发现而已，此时做什么都为时已晚。

管理者在给开发人员安排任务的时候常常询问他们需要多长时间完成，而开发人员常常说他们也不知道。开发人员并不是有意闪烁其词，也不是故意摆架子。他们只是不知道而已。坊间流传着一个关于软件开发的笑话：

开发者有三种状态：

已完成，
未开始，
快完事了。

1.8　评估未知

我们难以知晓某一任务需要的时间是因为我们从未做过。

有可能，事实上非常可能，我们少了一步。我们也许以为有些现成的代码会帮我们做些子任务，然后发现其实并没有，或者我们发现这一任务并非独一无二的，正好已经有一个库来完成我们所需的功能。最重要的是我们需要想透彻，而我们不可能像实际开发中那样有效地做到这点。

每时每刻，每天，每个月，每个项目，我们编写软件当中执行的任务都大有不同。当然，其中有些熟悉的事情我们每天都在做：设计、测试、编码等，也有些相似的技术我们每天都在用，也有处理问题的惯用方式。但是问题本身（以及它们对应的解决方案）常常和我们之前遇到过的大相径庭。

我知道有些公司（尤其是大公司）只给团队很少的时间去编码。很多时候还都是打着为了提高质量的旗号。

比如，花了很大力气用于编写内部设计文档并在开发过程中保持更新。在这样的开发过程中，所有的设计文档都一视同仁，但许多文档虽然曾经有用最后却被遗弃，显然不再有价值。

如果客户并没有花钱让我们编写这些额外的分析和设计文档，那么我们为什么还要这么做？答案是我们相信这些文档能帮助我们理解和表达问题，从而可以找到一个优秀的解决方案。但结果是我们常常花费大量时间在那些没有给客户带来价值的问题上。当我们关注于度量无意义的事情，诸如实际时间对比基于错误假设估计出的计划时间，那么我们就迈错了步子。

软件开发充满风险。少有能够顺利完成的，而且写出来的软件几乎马上就过时了。面对这不断增加的复杂度，传统的修复问题的方法是创立一个更好的流程。我们依赖流程告诉我们去做什么，保持我们步入正轨，保证我们诚实，确保我们的进度，等等。

这是瀑布模型开发背后的基本哲学。因为在开始的设计阶段之后便难以修改代码，我们禁止在设计完成之后再进行修改。因为测试消耗大量时间而且昂贵，所以我们等到项目的后期才统一进行。这种方式理论上行得通但是实际上却明显低效。在许多方面我们故意避免痛苦与困难，不是因为瀑布模型帮助我们构建更好的软件，而是因为它本身困难、耗时（至少比我们一开始预估的多）、花费更多金钱。

1.9　一个充满外行人的产业

软件产业被称为外行人的产业。不幸的是，这句话在很多层面都是正确的。

关于软件开发者应该具备什么样的知识体系并没有广泛接受的共识，开发者有大量不同的方式去解决问题。掌握一门编程语言并不能使你成为软件开发者，正如掌握一门自然语言并不能使你成为作家。

如果学生将来以软件开发为职业，许多计算机科学课程并未给他们做好准备。这些课程通常关注编程的数学特性，但这不是大部分软件开发者的工作内容。这就像在你成为画家之前要求你必须证明你能解复杂的微分方程。软件开发者有着不同的背景，通常用不同的方法解决问题。结果是，沟通设计思想并达成共识成为了难事。

科学和工程学的课程来自于广泛接受的标准和实践，软件产业却少有这类标准和实践。部分是因为我们解决的问题是如此天差地别，部分是因为这是一个年轻的产业。

我认识的最优秀的程序员都是自己独立解决问题。软件行业中重复发明轮子的事情有很多。

在土木工程中，也许对建筑美学有争论，但是在建造层面上却不会有。在给定结构尺寸和使用材料的情况下，需要相应的建筑材料和一套规定好的建造流程。建筑建造的规范依照这些原则实施。软件的情况却并非如此。

我认为我们永远也不会有一本如《土木工程指南》那样的《软件工程指南》。土木工程是——并非双关——坚如磐石的，正如软件工程是行云流水般抽象的。软件开发是在一个我们无法看到、听到、闻到或者摸到的虚拟领域中进行的。

这让我们中的大多数人难以理解。我们不习惯想象抽象的事物，我们也尚未理解软件开发的背景。我们无法像描述物理世界中的重力那样描述影响着虚拟世界的法则。

当代医学中，希波克拉底总结出了至关重要的前提："首先，勿做伤害。"医生的目标是治病救人，尽管有时候会为了拯救而损伤病人，比如锯掉感染的腿，最终目的还是挽救病人的生命。

但是软件开发中却没有对应的希波克拉底誓言——没有法则可以指导我们做出最好的选择。

软件开发流程中的每一步我们都会做出几乎无数个的选择。针对一定情况做出最佳权衡需要我们理解每个选项的所得所失。

为了解决本章开头提出的问题，无论是软件开发人员还是管理者都需要对软件开发的本质和软件随着时间变化达成共识，这样他们才能编写出可以更好应对变化的代码。我们必须填补基础认知方面的巨大空白。

软件产业面对的巨大挑战，对于那些准备正视它们且从中获益的人们来说也是巨大的机遇。这个年轻产业中的种种失败给我们指出了更好的方式，挑战着我们的基本假设，给我们展现了更优秀、更高效的工作方式。

所以我们必须要有勇气，不仅仅是提出一些困难的问题，而是要突破我们的固有观念、我们核心的信念和封闭的社区。我们必须共同回答这些问题，把软件开发变成更好、更高效、更可靠、更注重质量。

1.10 回顾

大多数软件开发和维护的方式“**有些不对劲**”，在传统的瀑布模型开发环境下，我们要等到最后才能看到一切是否正常运转，当bug出现的时候，它难以修复，非常耗时，以至于追踪修复的成本很高。

本章中心思想如下。

- 许多人不了解软件是如何构建的，软件又是如何演变成遗留代码（那些成本高且难以使用的软件）的，以及如何从一开始避免这样。
- 先分别开发功能再组装发布的方式效率低下。
- 传统的瀑布模型流程导致遗留代码的产生和繁衍。
- 软件工程尚未创建其核心的原则，也缺乏每个开发者都必须知道的通用知识体系。

瀑布模型促进了难以维护软件的产生。瀑布模型方法学在构造建筑方面很有成效，但是构造软件的时候先分别开发功能再组装发布是高风险、高成本的。传统的管理技巧通常不适用于软件工程，开发者也没有可以共享的通用知识体系。

我看到了这些想法在我接触的项目中的负面影响，但不知道其他项目是如何深受其害的。接下来，我们会将软件产业视为一个整体看看它的情况如何。

第 2 章

逃出混乱

如果你和我一样，那你应该也知道现如今多数软件的开发与维护方式有着种种问题，但是却找不到问题的症结所在。没有这一数据，你可能无法说服你的团队成员或管理者，让他们意识到遗留代码不仅仅是一个商业成本问题，而且是一个真正迫在眉睫的危机。如果想避免重蹈那些失败项目的覆辙，首先需要意识到都有哪些关键问题导致了项目的失败。

软件产业总体来说是一个尚未被充分探索的行业，尤其是考虑到世界范围内有多少的其他产业和软件息息相关。斯坦迪什咨询集团（Standish Group）试图对这个年轻、复杂而又混乱的软件产业一问究竟。尽管他们承认数据可能不够全面，但的确可以让任何一位做过几天软件开发的人都能够清楚意识到，我们这个行业的成功率其实并不算很高。

我亲历过一些项目的苦苦挣扎甚至失败，但我从未想过整个行业都处在危机之中，每年都因为破败不堪的软件开发流程损失数以百亿计的美元。作为一个给全世界最大的软件公司做过咨询的人，我想知道我的经历是否属于典型。所以我把目光投向了行业中规模最大、最受关注且被广泛引用的研究：**斯坦迪什咨询集团的“混乱报告”**。

2.1 混乱报告

斯坦迪什咨询集团[①]是一家专注软件产业的研究机构。他们最著名的研究名叫“混乱研究”（CHAOS Study），观察了大量的软开发项目并通过不同指标评估其成功与否。此项研究囊括了3.4万个软件项目，从套装软件和操作系统到定制化应用和嵌入式系统。连续十年的研究囊括了大量不同的软件项目和不同的参与者。每年停止跟踪3400个十年前开始的项目，同时开始跟踪3400个新项目。

斯坦迪什咨询集团给365个参与者发送问卷，覆盖了8380个应用，为的是把项目归为三类。

① http://www.standishgroup.com

2.1.1 成功的

斯坦迪什咨询集团对成功的软件项目的定义是："按时完成，没超过预算，所有的功能和性能与预先定义的一样。"

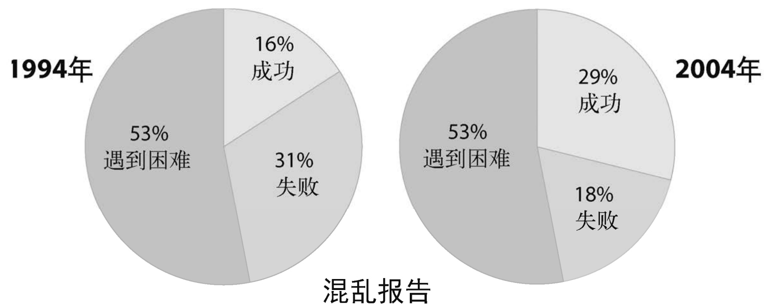
2.1.2 遇到困难的

遇到困难的项目不太容易界定。这些项目虽然完成了，但是总会有些妥协，不是预算上的（成本过多），就是时间上的（延迟交付），或者交付较少的功能，或者性能不如预期。

2.1.3 失败的（有缺陷的）

失败的项目是那些被取消的、不曾见天日的项目。通常，项目由于开发者以外的原因被取消或者失败。一个项目可能因种种原因导致失败：资金不足、市场变化、公司策略变化，等等。

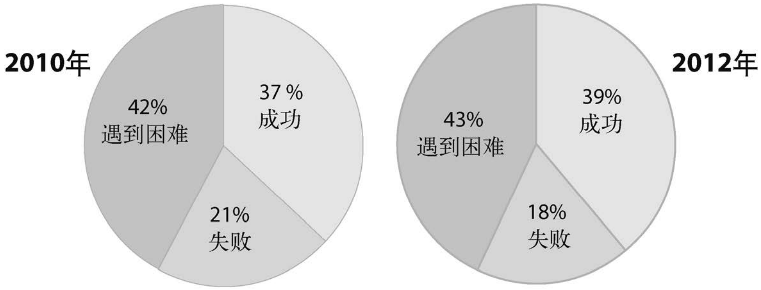
在1994年，混乱研究[①]显示3.4万个项目中有16%是成功的，53%遇到了困难，31%失败了。十年之后的2004 年[②]，29%的项目成功了，只有18%的项目失败了。



混乱报告

在2010年，混乱研究[③]显示有37%的项目成功，42%的项目遇到了困难，21%的项目失败。仅仅两年之后，在2012年[④]则是39%的项目成功，只有18%的项目失败。

① http://www.projectsmart.co.uk/docs/chaos-report.pdf
② http://www.infoq.com/articles/Interview-Johnson-Standish-CHAOS
③ http://www.versionone.com/assets/img/files/ChaosManifest_2011.pdf
④ http://www.versionone.com/assets/img/files/CHAOSManifesto2013.pdf



虽然在这些年中有着长足的进步，但是软件项目成功的比例依然不高，仅有不到三分之一的项目能成功，至少斯坦迪什咨询集团是这么说的。

2004年成功率的提高，我相信很大一部分要归功于软件产业的成熟和敏捷方法论被越来越多地采用。但是，虽然有“敏捷”这样的新想法，依然只有三分之一的项目能够成功。

我曾经为一些非常庞大的组织效力过，他们中大多数（但并不是所有的）组织的成功率更加低。那些并非单纯从事软件开发的公司甚至有时只有5%左右的成功率。

业界对这项研究非常关注，很多人翘首以待。但是这项报告发布后不久，期待就变成了质疑，不是对这项研究的结果，而是对这项研究本身。

2.2　驳斥斯坦迪什咨询集团

所有的研究都应该证明其客观性，但是斯坦迪什咨询集团采用的对成功的定义非常不精确：“按时完成的，没超过预算，所有的功能和性能与预先定义的一样。”

“预先定义”对于预算、时间和功能来说，最多不过是最合理的猜测。而且也没有关于软件上市之后状况的数据。

初始的定义依赖于我们**精确**了解需要这个软件做什么，**精确**了解如何让其工作，**精确**了解需要多长时间，**精确**了解期间公司会遇到什么样的挑战……而没有给更好的想法留出任何生存空间。

斯坦迪什咨询集团对成功的定义刚好是个关于失败的食谱。它建立在一个荒谬的概念上，以为我们可以事先预知一切，但我们连精确了解其中任意一项都是不可能的，更别说全部了。更糟糕的是，它假设我们在每个项目开始的时候知道每件事情的最佳做法，以至于没有留下任何时间去实验、讨论、修改、重新思考等，没时间想出更好的主意。

在论文《混乱报告数据的起起落落》[①]中，信息技术产业研究员 J. 劳伦斯・伊夫琳斯和克里

① Eveleens, J. Laurenz, and Verhoef, Chris. “The Rise and Fall of the Chaos Report Figures,” *IEEE Software*, 2010, 27(1) http://www.computer.org/csdl/mags/so/2010/01/mso2010010030-abs.html

斯·弗霍夫说明了，斯坦迪什咨询集团对“成功的”和“遇到困难的”项目的定义具有误导性，是片面的，导致了对实际评估的曲解而得出无意义的数据。

混乱研究真正度量的是项目管理团队对时间成本预估的能力，以及推动需求的能力。正如伊夫琳斯和弗霍夫指出的那样，斯坦迪什咨询集团的定义并未真正考虑软件开发的真实场景，如可用性、收益和用户满意度。

实现最初的目标其实更像是对于“失败的”定义，因为它意味着我们没有弄清楚客户真正想要的东西，也不知道如何构建出优于一开始所设定的软件。我们只是单纯为了把事情做完而开工：按时交付、符合预算、仅仅做初始需求所列出的功能。

按照斯坦迪什咨询集团的定义，如下这样的项目也会被归为“成功”之列。

- 程序一个月后崩溃。
- 用户想要添加一个无法完成的简单功能，需要大量的金钱和时间投入，或者引入诸多新bug。
- 糟糕的代码在项目中成年累月的堆积，遗留代码造成了大量的“技术债”。
- 这是客户最后一次从我们这里采购。

这也许意味着，多数混乱报告中被标记为“成功的”项目也许并非真正成功了。

由此类推，被斯坦迪什咨询集团归为“遇到困难的”项目也许并未达到其预期的功能、时间表或者预算，也许获得了市场上的成功。

我们从混乱报告中可以了解到的唯一一件事就是，那些在开发过程中被砍掉的“失败的”项目，从未发布过。

这些简单的标准永远不会深究为什么项目会失败，或者为什么项目会“遇到困难”。所以最终的结果就是声称：“由于种种原因，我们总是搞砸。”

我可不想就此罢休。

2.3 项目为何会失败

我们不喜欢讨论失败。失败意味着我们有些事情没有搞清楚。但是失败确实存在，可能软件行业比其他产业更加深受其害。虽然混乱研究所用的方法受到了质疑，但是不可否认软件开发行业确实有很大的改进空间。不管数据是如何收集的，任何在这个行业工作过一段时间的人都会意识到，一个新项目失败的概率比成功要大。这让软件开发在很多公司成了一个“不靠谱”的提议，而且失败通常会引来广泛的关注。

并非只是一些预算紧张的创业公司容易失败。全球范围内的大公司以及预算充足的组织也面临同样的问题。

1994年，美国联邦航空管理局在花费了纳税人26亿美元后，取消了一项升级空中交通管制系统的关键项目。2004年，福特汽车公司弃置了花费4亿美元的新采购系统。根据一篇发表在2014年4月的《华盛顿时报》上的文章①，修复问题频发的奥巴马医保网站的花费将达到1.21亿美元，甚至比一开始创建网站的费用还高出2730万美元。

而且这类事情远非十年一遇。

按照斯坦迪什咨询集团的说法，一个失败的（有缺陷的）项目“是在项目开发周期中被砍掉的”。这样的定义避免了对“成功的”与“遇到困难的”项目定义的歧义，明确指出项目是被砍掉的。

但是许多项目被砍并非是开发层面的失败导致的，而更多是商业重心的调整和多变的市场需求导致的。无论是什么导致了项目被砍，通常都不是开发环节所能控制的，然而项目“遇到困难”却常常是因为糟糕的编程方式和其他技术层面的原因。

为了表达清楚，我们将这一复杂的问题分为三个导致低成功率的核心因素：

(1) 代码变更
(2) bug修复
(3) 复杂度控制

2.3.1　情况发生了改变

软件不会自己更新自己，它保持一开始被编写出来的样子。Windows 7并非由于自然选择而进化成Windows 8。但是如果软件投入使用就必然需要**修改**，而且需要有人亲力亲为去修改它——这是件好事。这意味着人们发现了从软件中获取更大价值的方式。只有当软件没人使用的时候才不需要修改。

向软件中添加功能是一个常见的需求，但直到现在我们都没有找到一个稳健的执行方式。通常维护代码的团队并非一开始开发的团队，即使是一开始编写代码的开发者被要求扩展其编写的代码的时候，他们也常常难以回想起当时设计程序的方式。

多数情况下，开发者阅读代码要比编写代码花费更多的时间，所以如果需要大范围的修改，他们更倾向于重写一段段代码，而不是费力气去弄明白它们。在需要基于现有代码增加功能的时候，与其花时间弄清楚现有的设计，许多团队直接强行修改，让系统的整体质量下降，使以后的扩展难上加难。

在现有的软件中添加功能的成本可能会极高。这是因为多数的软件并未针对可扩展性做过相

① Howell Jr., Tom, and Dinan, Stephen. “Price of fixing, upgrading Obamacare website rises to $121 million,” *The Washington Times*, April 29, 2014. http://www.washingtontimes.com/news/2014/apr/29/obamacare-website-fix-will-cost-feds-121-million/?page=all

应设计，往往在添加功能前需要重新设计。这可能会有风险而且代价很高，所以开发者最终选择在代码上堆叠代码。软件错综复杂的本质使其难以在不引发连锁反应的情况下增加功能。新的功能引入新的bug，从而导致另外一个bug，然后一个接着一个，如此往复。在没有针对可扩展性做过设计的软件上进行扩展无异于雪上加霜。

更改代码，即使是很小的更改，也常常需要对整个系统进行重新测试。对于大多数程序来说，这意味着需要大量的人力去重新执行所有的测试，以保证新功能增加的时候没有出错。

如果这听上去势如累卵，那么你基本上是正确的。开发者编写的代码**无法修改**是再常见不过的了。

2.3.2　bug 泛滥成灾

软件时常因bug失败，而修复bug又代价高昂。它们可以让开发成本增长一个数量级，使项目停滞，对系统造成损害。

软件开发者通常不会在修复bug上花费太多时间。虽然有些十分顽固的bug需要大量时间修改，但是大多数bug是微不足道的。然而**找出**这些微不足道的bug却绝非易事。

最新版本的Mac OS X约有8500万行代码[①]，文本长度相当于近1200部《战争与和平》。想象一下，如果需要在世界上最长小说之一的1200倍长度的文本中找出一个拼写错误意味着什么。

然而一个拼写错误足以让一个8500万行代码的程序崩溃。

所以真正的问题是，我们如何让bug更容易被**找到**？或者更进一步，我们如何**从源头避免bug的产生**？

当我最终意识到程序中的bug是出自我手时，会觉得意外，而且有点难为情。开发者写bug的方式和写代码一样，只不过没人花钱雇他们写bug。

bug种类很多：小到能通过编译的拼写错误，大到系统级的设计缺陷，统统都是。

bug常常仅仅是冰山一角，修复一个问题可能会导致更多的问题浮出水面。软件内部常常以盘根错节的方式编写，所以修复bug可能很快变成打地鼠一样的体验，一个看似简单只需要几分钟的修复过程可能变为系统中无穷无尽的修改。

作为开发者，我和bug打过很多年的交道，但是直到最近才开始意识到它们的本质：bug是软件开发流程中的缺陷。

和真正的昆虫一样，软件的bug需要合适的条件才能生存。

---

① Information Is Beautiful. “Codebases: Millions of lines of code,” v 0.71, October 30, 2013. http://www.informationisbeautiful.net/visualizations/million-lines-of-code/

2.3.3　复杂性危机

2002年，美国国家标准与技术研究所（NIST）称：①

> 软件之所以充满了错误，是因为其不断增长的复杂性。软件产品的规模不再是数以千计的代码行，而是以百万计。软件开发者已经在排查与修复的工作上花费将近80%的开发成本，尽管如此，依然鲜有其他类型的产品如软件一样带着如此多的高级错误交付。

80%的开发成本花在了“排查与修复”上！这意味着只有20%的预算创造了价值。显然是我们的急功近利造成了如此多的问题，以至于需要花费80%的成本回过头去修复。难怪开发者只有那么少的时间！但是这番窘境的解决之道绝非是像“一开始就把事情做好”这样的陈词滥调。正如我们后面会看到的，真正的问题和解决方案要复杂得多。

多数软件的编写方式让它们阅读起来比编写更困难。软件里面充满了依赖——一段代码依赖于另一段，而那段代码又依赖于其他代码，直至整个系统都互相纠结成了一大团——加上每个开发者写代码的时候都或多或少会“重复发明轮子”，所以很难或者完全没法预知某项功能究竟会被如何建模。

“混乱报告”引出了另外一个值得关注的事实：用户实际使用的软件功能的占比。斯坦迪什咨询集团的研究表明，在超过3.4万个软件项目中，只有20%的功能被经常使用，45%的功能**从未**被使用过。②

从未使用的功能并非指那些类似恢复备份这样的功能。我们并不会每天都恢复备份。很少使用绝非不重要。从未使用指的是从来、**从来**就没有使用过。

编写某项功能的开发者可能在调试的时候执行过，但是除此之外从来没有任何客户见过它。

为什么会开发一些没人想要的功能呢？部分原因是由于市场营销。也许在一个功能清单上看着很好，也许客户以为他们想要所以加入到了冗长的需求列表中，他们知道，在瀑布模型的项目中，只有一次列出需求的机会。

但是开发者也需要为过度开发负责。我们总以为，**我刚好正在编写这部分的代码，这个功能也花不了几分钟——举手之劳而已**。但是我们并没有意识到增加功能也许是举手之劳，但是将来的维护却要一小笔花费。

我们把“一小笔”“一小笔”的花费加起来，一笔又一笔，在整个软件产业范围内，开发、维护、扩展软件的成本就成了一大笔。但是我们并未止步于此——我们同样也将大笔的开销累积，直到如参议员埃弗里特·德克森所说的那样：“这里十几亿，那里十几亿，很快你就面临着天文数字了。”

---

① National Institute of Standards and Technology. “Software Errors Cost U.S. Economy $59.5 Billion Annually: NIST Assesses Technical Needs of Industry to Improve Software Testing,” June 28, 2002. http://www.abeacha.com/NIST_press_release_bugs_cost.htm

② Fowler, Martin. “Build Only the Features You Need,” July 2, 2002. http://martinfowler.com/articles/xp2002.html

2.4　失败的代价

许多已发布的关于软件产业的研究都是十年或者更久之前的，而且尚未被新的研究所取代。但是，这些研究中的许多发现至今依然相关。

这些年间我曾询问过参加我课程的数以千计的专业软件开发者，他们感觉软件项目的成功率有多少。我得到的答案是5%~30%。

虽然我们尚未讨论这个报告，但是这个非正式的数据和“混乱研究”的发现相差无几。开发者的回答反映了众所周知的一些东西，无论是否有严格的统计数据做支撑：

我们的产业距离有效和高效还有很长的路要走。

软件开发项目如果失败多于成功，对产业来说意味着什么，对客户和开发者来说又意味着什么。我们可以计算出这些在失败的软件开发期间损失的时间和资源，但是我们没法计算错过的机遇。

丹佛机场由于行李处理系统的问题而延迟开放造成了5.6亿美元的损失[①]，像类似的事情我们都听说过。我们不希望听到小的失败时常发生，然后迅速堆积起来。

当今软件已经差不多成为各行各业的心脏，体现在它几乎帮助我们规划与实施每件事情。越来越多的公司，无论他们本来是做什么生意的，都已经开始做软件生意了。

2.4.1　这里十几亿，那里十几亿

美国国家标准与技术研究所2002年的报告“软件测试基础性的缺乏对经济的冲击”[②]发现，软件缺陷对美国经济每年造成近600亿美元的损失。

让我们来展示一下600亿美元意味着什么。

- 600亿美元比全球180个经济体中70%的国民生产总值还高。[③]
- 如果将Facebook的创始人马克・扎克伯格和亚马逊的创始人杰夫・贝佐斯的身价加起来，差不多有600亿美元。[④]

而且是**每年**600亿美元的损失。

---

① Calleam Consulting “Case Study – Denver International Airport Baggage Handling System – An illustration of ineffectual decision making.”(2008) http://calleam.com/WTPF/wp-content/uploads/articles/DIABaggage.pdf

② National Institute of Standards and Technology. “The Economic Impacts of Inadequate Infrastructure for Software Testing,” May 2002. http://www.nist.gov/director/planning/upload/report02-3.pdf

③ Serafin, Tatiana. “Just How Much Is $60 Billion?” Forbes (blog), June 2006. http://www.forbes.com/2006/06/27/billion-donation-gates-cz_ts_0627buffett.html

④ Forbes 400. http://www.forbes.com/forbes-400/list/#tab:overall

仅仅是在美国。

事实上，这些数字很难说得上完全准确，但是因为这个问题对我们要讨论的其他问题来说是如此重要，让我们先看一下对软件开发失败损失进行量化的其他尝试。

2.4.2　不同的研究，同样的危机

在《软件项目的失败造成十亿美元级别的损失：更好的评估与计划可以用于改善》[①]中，作者丹·格罗拉斯的研究发现“基本上认同”软件开发失败的损失在“每年500亿到800亿美元之间”。如果这是真的，我们每年损失少则是福特汽车公司的资产总值，多则为中国石化集团（全球第五大公司）的资产总值。

罗杰·塞欣斯在他非常有影响力的白皮书《信息技术复杂性危机：危险与机遇》[②]中，敲响了警钟：

> 信息技术产业面临着崩溃。激增的信息技术工程失败在未来会失控，没有国家或者企业可以幸免。信息技术项目失败会在侵蚀美国利润的同时，也影响着澳大利亚的经济。信息技术项目的失败在私人产业、公共事业和非营利事业上同样猖獗。没有哪个地方是安全的。没有哪个产业受到保护。没有哪个行业能够幸免。这是危险的，也是真实的。

这可真是一大堆的坏消息。我们在半数以上的情况下会搞砸，而且还花费大量金钱。

即使假设美国国家标准与技术研究所给出的600亿美元的数字有着50%的偏差，那么每年小十一位数的损失比每年大十一位数的损失更容易接受吗？每位管理者都会停下来问一句：“我们公司给这个十一位数贡献了多少？”

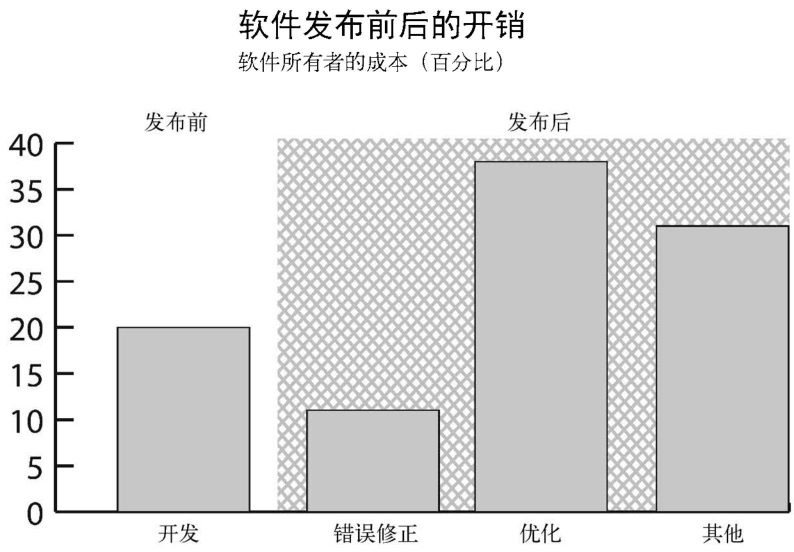
这些软件开发的大量成本和软件维护的成本比起来简直小巫见大巫。80%的软件开销花费在初始发布之后[③]，维护软件时期的花费可能比开发时期的花费高出5倍之多。正如下图所示，差不多60%的成本花在了优化上，17%花在了错误修正上。

---

① Galorath, Dan. “Software Project Failure Costs Billions: Better Estimation & Planning Can Help.” *(blog)* June 7, 2012. http://galorath.com/wp/software-project-failure-costs-billions-better-estimation-planning-can-help.php

② http://simplearchitectures.blogspot.com/2009/11/it-complexity-crisis-danger-and.html

③ Glass, Robert L. “Frequently Forgotten Fundamental Facts about Software Engineering,” *IEEE Software* archive, 18(3), May 2001, pp. 111–112 http://dl.acm.org/citation.cfm?id=626281



为何软件维护成本如此之高？简单来说，我们没有对可维护性给予足够的重视，让可维护性成为项目的首要目标之一，所以构建的软件充满了风险而且修改的成本很高，没人知道经济上的影响有多大。

即使无法确定如何度量成功与失败，也不知道如何跟踪整个产业的每个角落损失的金钱，但我们都是聪明人，当愿意承认“可以做到更好”的时候，我们至少开始去尝试了。

2.5 总结

这一章清晰地展示了维护遗留软件的巨大成本。即使对研究方法和百亿美元级别的数字存疑，也可以明显看出这个行业浪费了大量的金钱、精力和客户的美好愿望。

本章中心思想如下。

- 低效的软件构建方式每年造成大量损失，我们可以直面也必须直面这个危机。
- 广受引用的软件产业研究“混乱报告”，虽然有着基础性的缺陷，但是它得出的结论——我们的产业前路漫漫——是正确的。
- 业界主要的研究显示，仅仅是在美国，不健全的软件开发流程造成每年损失数百亿美元。
- 遗留代码是个全球性的问题，而且是我们共同造成的，所以我们有责任解决它。

斯坦迪什咨询集团的“混乱报告”关注软件项目的成功率。它表明多数项目并不成功，但是对“成功”的定义却十分草率，所以这份报告并没有多大用处。尽管如此，其他的报告同样确认

糟糕的软件开发实践给行业带来每年数百亿美元的损失。

虽然看起来多数的软件项目注定失败，但是有些软件项目却获得了成功，而且其中许多采取了不同的开发方式。我开始关注其他的软件开发方式。让我们看看一些来自于聪明人的、对软件开发有益的新想法。

第 3 章

聪明人，新想法

在千禧年到来之际，一群成功的软件开发者意识到，对于软件开发流程来说，少即是多。从不同的角度出发，他们努力找到一个轻量级的软件开发流程。一开始，他们称其为“轻量级软件开发流程”，但是担心这样的名字不能引起足够的重视，于是改名为“敏捷软件开发流程”。

公平来说，“敏捷”来源于W. Edwards Deming和他的“精益”概念。然而直到敏捷概念引入后，Mary Poppendieck才将“精益”原则带到了软件开发中。根据“敏捷”背后的“极限编程”哲学，Ken Schwaber和Mike Beedle合著了《Scrum敏捷软件开发》[SB01]，帮助团队在工作中更迅速地实现极限编程。

一切都朝着正确的方向前进，而且正如我们在“混乱报告”的统计数据和其他资料中看到的，“敏捷”和软件产业中猖獗的低效展开了斗争。无论如何，我们都应深入了解这些“新”想法，去看看为什么如此多的聪明人直到十多年后才想到优化改进的方案，他们又是如何做到的。

3.1 走进敏捷

当杰出的统计学家W. Edwards Deming于1950年到访日本的时候，正如你们想象中的那样，他看到的是一个支离破碎的国家。第二次世界大战的硝烟刚刚散去五年，日本的大部分基础设施依旧处于瘫痪状态。日本的重建说好听些是举步维艰，说不好听些则是白日做梦。作为最初被派往日本帮助进行1951年日本人口调查的团队中的一员，Deming开始和一些顶尖的工程师与管理者一同工作，不单单是要把日本的经济和工业恢复到战前水平，而且要使其跃上一个没人想象过的新高度。和其他成员一样，W. Edwards Deming为日本战后的“经济奇迹”做出了贡献。

但这并非奇迹。

Deming带来的观念让日本企业将关注点放在了持续质量改进这一概念上。这需要确立一组质量优先的标准以及各级组织专注于生产质量和用户价值的最大化的承诺。Deming的想法被丰田公司采纳，最终发展成“精益”理念，这个理念显然在丰田起到了很好的作用。

“精益”是一个在过去二十多年中被证实的方法，丰田在美国建厂后该方法被带回了美国。现在，美国有许多丰田的工厂，它们在截然不同的美国工作环境下，依然令人难以置信地高效运

转。事实上，起决定性作用的是流程而非人力。丰田有一套十分出色的流程，它的成功带动了汽车产业改革，也同样影响了许多其他产业。

Deming关注的是**浪费**，以及如何消除这些浪费。在制造业中，库存占据了仓储空间，所以被视为浪费。这意味着库存不仅仅冻结了资产，而且每天都会产生持续的仓储花费。直到将产品发送给客户，你才能将库存产生的负收入变为盈利。这很容易理解，而且可以回溯到20世纪20年代Henry Ford提出的即时供应链理论。

但什么是软件中的浪费呢，和汽车不一样，软件不需要钢铁、轮胎和其他供应，情况又如何呢？

“精益”理论认为，软件的浪费是所有已经开展但尚未完成的任务，是半成品（Work-In-Progress，WIP）。我可以进一步引申，任何不是软件的东西，或者没有给客户产生直接价值的东西，都可以视为浪费。

这种理念是包括极限编程、Scrum和精益在内的敏捷软件开发方法论所一致认同的。虽然本书不是关于某单一方法论的，但是许多后面要讨论的实践方法都是在2001年著名的“敏捷宣言”中提出的。这个宣言表示：“我们一直在实践中探寻更好的软件开发方法，身体力行的同时也帮助他人。”[①]

“通过持续不断地及早交付有价值的软件使客户满意”，这一承诺是敏捷流程的核心。换句话说，敏捷理论摒弃了用增加流程来保证质量的方式，建议流程更加精简，好让开发者有更多的时间实施更切实际的工程实践。敏捷理论引入了一些技术性实践，如测试驱动开发和结对编程，这些实践有助于创建可修改的，更容易部署、维护和扩展的软件。

3.2 小即是好

长跑运动员们跑步的时候会给自己设立相对短程的目标。“我要坚持到那个路灯”感觉更可行，也更简单，尤其是和42千米的马拉松相比而言。到达那个路灯后，再设立一个新的目标：那棵大树。那个红色的汽车。诸如此类。

软件开发者并不是总能确定——事实上我们几乎从未能够确定——一个项目需要多长时间。原因有许多，绝不是因为懒惰。在很多方面，软件设计这个专业几乎刚刚从起跑线出发，而且我们不确定是否能够完成整个马拉松。我们从起跑线上看不到终点，甚至不知道到底需要跑多长时间。终点线也许在42千米外，也许在21千米外，也许在84千米外。或者我们只知道终点在哪，但是没有地图，没有预先订制的路线能够保证我们顺利到达那里。

但是如果将注意力从整个赛程转移到其中的一小部分，比如这两周的而不是一整年的开发工作，会怎么样呢？这样的话，和长跑运动员一样，我们可以对一小段赛程负责，一步一个脚印。

① http://agilemanifesto.org/

我们跑了多远？是否需要加快速度或者重新评估？是否可以提前完成？

本书第二部分提到的很多想法都来自敏捷运动。这些想法是由那些聪明人最先提出的，他们很早之前就看到了瀑布模型的缺陷，以及对软件开发者这个职业产生的负面影响。但是无论这些敏捷背后的人有多聪明，无论这些想法有多出色，前路依旧漫漫。

3.3 实现敏捷

敏捷2001年就已经诞生了，但是在软件行业中依然未被广泛理解。许多组织只实施了一个或者几个简单的敏捷实践，诸如"站立式会议"和"两周冲刺"，然后就声称自己已经"敏捷"了。许多Scrum团队知道有一个"产品负责人"负责控制被称为**时间盒子**的时间线。这些都是重要的实践，但是仅当使用得充分、得当且和其他实践配合的时候才有价值。

这并非敏捷和瀑布模型的对决。瀑布模型的很多过程很容易带入敏捷实践之中。比如，如果需求收集变得非常复杂，必须要写下来以便日后阅读和讲解，那么就可能导致严重的低效和bug。

保留独立的质量保证过程，让开发者把软件交给测试人员验证，这也是昂贵且低效的。保留这样的行为然后实施"两周冲刺"和"站立式会议"并不会取得多大改进。

我更倾向于那些强调软件开发中的原动力的实践，而非简单地贴上"瀑布模型"或"敏捷"的标签。本书并非说瀑布模型是糟糕的而敏捷是好的。核心问题远远没有这么简单。

以我的经验来说，在敏捷环境中开发软件的诸多好处之一是，敏捷有助于建立一个发现式的过程，团队持续收到反馈并从中学习。敏捷需求（在敏捷中叫作故事）并不是取代了那些在瀑布模型中要求的长而详尽的说明文档。故事的目的是引发对话——它要求并且鼓励软件开发者和产品负责人进行有意义的交流。从这类交流中开发者能够了解到实现需求所需的东西。

单纯去掉需求说明文档而没有用软件开发者和产品负责人之间的交流取代，并不是敏捷的初衷。如果产品负责人成天所做的就是从客户那儿拿到需求说明再进行细化之后一路维护文档，那不过是瀑布模型加上一个产品负责人而已。那不是敏捷。

我们需要深入其中，去了解敏捷中隐含的精髓，并不仅仅是有一个产品负责人然后抛弃所有文档，而是要真正地将对话的主题从**如何去做**变为**做什么**和**为什么这么做**。

举例来说，实践"小批次构建"的核心目的是为了将任务从开始到完成的周期缩到最短，更小的任务可以更快地完成。许多团队虽然进行迭代开发，但是他们的工作在一个列队里面等着质量保证团队进行发布前的测试。他们把工作完成了99%，但是还不够好。他们的代码中隐藏着未知数量的风险，直到集成和测试**之后**才能发现。

当人们明白"小批次构建"是为了尽可能快地完成任务、限制半成品存在的时候，他们才能更有效地应用这些实践，进而看到更大的收益。

同样，"时间盒子"也可能被误解和误用。在极限编程中，将大问题用时间盒子分解为小问

题的方法叫作“迭代”，在Scrum里面叫作“冲刺”。但是我对这两个名称都不是很喜欢。它们容易给人以错误的印象。

敏捷和Scrum并不是急于求成而是循序渐进。敏捷本质上就是“范围控制”——限制我们工作的内容——我们用时间盒子来度量仅仅是为了让人们习惯于范围控制。

时间盒子是说，“我们要在固定的时间内做这个工作”，通常是一个非常短的时间段，在Scrum里面差不多是1~4周，比较常见的是为期两周的一个迭代或者冲刺。

关键是，我们应该以更小的**范围**或者**工作**单元来思考，而不是时间长度。我们将大任务分解为更小的工作单元，依然可以产出可观测的结果。工作单元越小越好——小的任务更容易预估、实现和验证。

Scrum被当作一个管理软件开发的方法论而忽视了其中来自极限编程的技术实践。 Scrum鼓励团队自我管理，听起来是个好主意，但是仅仅告诉开发者自我管理并不能保证他们采用那些实践，而那些实践有助于他们关注代码质量，编写出可维护的软件。

仅仅是回过头在几周前编写的代码中添加功能，就足以驱使开发者开始编写更易维护的代码了。但是当这些技术实践被忽视、误用，或者纯粹被误解的时候，团队到头来口头上说“我们Scrum了”，实践上却依然是在繁重的需求、测试滞后的瀑布模型环境下工作。他们虽然以更小的批次编码，但是依然充满了依赖，使得代码难以使用，并且bug在最后才得以发现。

我见过Scrum团队虽然立竿见影提高了效率，但是在项目进行了四五年后，他们积累的技术债和糟糕的代码让他们几乎无法工作。最终，他们不得不承认开发实践的不足之处，从他们自己创建的低质量代码中挖出一条路来才能继续有效开展工作。

我见过采用Scrum和极限编程的团队彻底的失败。我见过团队使用瀑布模型并且成功了。每个方法都有其优点和短板。我们必须先明确每个实践的作用才能正确使用它们。

我们讨论过，影响软件工程的因素和我们现实中的其他经验大不相同。即便是电子工程和机械工程也都是基于物理的。但是软件不遵循物理法则，时常难以体会，对其难有真正详尽的认识。

近乎所有物理世界的事物都是有容错性的。有生命的和无生命的系统都有着很大的弹性。但软件是世界上最脆弱的事物。一个错误的比特可以导致灾难性的系统故障。因为这一事实，我们必须用一种可验证的方式构建系统。

软件开发是非常反直觉的。引发了制造业革命的质量控制标准在应用到软件项目时遭遇了全面的失败。我们在工业革命时期学到的东西在软件上毫无意义。软件是完全不同的生物。

3.4　艺术与技能的平衡

软件开发是一个复杂且多样化的领域，包含了许多的技巧和能力。开发者必须利用不同的技

术，因为他们今天面对的和昨天面对的是完全不同的问题，而且明天也是一样。正因如此，开发者需要大量的技能用于处理不同的问题。正如木匠用工具箱把常用的工具带在手边一样，开发者需要许多随时可用的**智力**工具来处理大量未知的软件问题。

事实上，任何人都能够写出一个简单的程序让计算机做些任务——这其实是一项非常容易学习的技能。这就是软件开发的**技艺**：一组可以学习的技能，进而通过实践不断精进。技艺是规范的。就是在学校里面教授的那些。你也许在中学就学习了遣词造句所需的一切，但是并不能让你成为赫尔曼・黑塞[①]、杰罗姆・大卫・塞林格[②]或者斯蒂芬妮・梅尔[③]。在软件开发的技艺中你需要一些基础训练，但是那仅仅是个开始。

不幸的是，许多开发者在学校里面学到的技能是过时的，而且助长了难以维护、难以扩展代码的产生。开发者也许能编写并执行一个程序，但是回过头去扩展代码却是一个艰难而冒险的提议。

在敏捷软件开发中，开发者时常回到之前的代码当中扩展代码行为，这需要大量的技术性实践和关于软件的思维方式，而这正是大多数开发者所不具备的。

如果在软件行业中浸淫许久，一个开发者也许会有机会学到多种领域的知识，比如视频压缩、外汇储蓄、自动驾驶船舶、计量经济学、图像和图形处理、遥感技术、信号处理、大数据等。每个项目都是一个学习其他产业知识并且解决一些特定问题的机会。这些特定的问题需要特定的解决方案。

开发软件需要许多的技能和能力，也就是技艺，但是无论学到多少技能都没办法解决**所有**问题。软件开发是为数不多的同时使用左脑（客观、逻辑……技能）和右脑（主观、创造力……艺术）的领域之一。听我这样说，许多人会觉得很奇怪。他们想象中的编程是完全理性的——全都是算法——但是编程的人都知道：需要创造力和想象力才能编写出出色的代码。

但是软件开发依然是一个年轻的职业，软件行业也刚刚开始被其他行业所接受。

3.5　敏捷跨越鸿沟

Geoffrey A. Moore在他的《跨越鸿沟》[④]中提到新产品的“技术采用周期”。他描述了对任何创新产品接受情况截然不同的五个群体。

- **创新者**首批采用新技术。
- 受到创新者成功经验的启发，**早期实践者**是下一批采用新技术的。

① 赫尔曼・黑塞（Hermann Hesse，1877—1962），德国作家、诗人。出生在德国，1919 年迁居瑞士，1923 年 46 岁时入瑞士籍。1946 年获诺贝尔文学奖。代表作《荒原狼》《东方之旅》《玻璃球游戏》。——编者注

② 杰罗姆・大卫・塞林格（J. D. Salinger，1919—2010），美国作家，代表作《麦田里的守望者》。——编者注

③ 斯蒂芬妮・梅尔（Stephenie Meyer，1973—　），美国作家，代表作《宿主》《暮光之城》系列。——编者注

④ Moore, Geoffrey A. Crossing the Chasm. New York: HarperBusiness (1991)

- 当出现了很多的操作指南、技术变得更加易用的时候，**多数派先行者**开始加入了。
- 在多数派先行者的帮助下，技术变为主流，**多数派后行者**也加入了。
- 最后，那些**滞后者**在别无选择的情况下才加入。

我们不仅可以在每个创新产品出现的时候见证这个“技术采用周期”，同样的采用周期在任何创新产生的时候都重复上演。Moore将这个周期的中点，一项创新由早期实践者传入多数派先行者的时间，称为“跨越鸿沟”。

敏捷已经被早期实践者所接受，并且已经开始渗透到多数派先行者中。敏捷**正在**跨越鸿沟。也许等到十五年后或者更久，敏捷依然没有**完全**跨越过去，即便是已经跨过了鸿沟前方依然充满了挑战。

敏捷中的一些方面，比如说极限编程中的一些技术实践，依旧处在创新者阶段。**阶段性的创新**被阶段性地采用并不罕见。新技术常常在人们感觉最稳妥的时候才会被采用。这就是为什么那些简单常见的、但是没那么有价值的实践通常被首先采用的原因。正如我们在下一章看到的，直到我们都理解这些实践背后的原则之前，敏捷都不能称为“常态”。

3.6　追求技术卓越

敏捷宣言的一些创始人对采用敏捷的看法并非秘密。在敏捷宣言十年重聚的时候，Jeff Sutherland说采用敏捷的第一成功要素是**追求技术卓越**。

尽管敏捷宣言称“坚持不懈地追求技术卓越和良好设计，敏捷能力由此增强”，但很多敏捷宣言作者后来觉得强调技术卓越的力度还不够，进而提出“追求技术卓越在未来十年内将是头等大事”。[①]

他们所谓的“技术卓越”到底是什么？鉴于现今大多数的软件都难以使用，所以可以说当前大多数的开发者并不真正了解“技术卓越”是什么。

因为软件是抽象的，所以有时候难以捉摸，尤其是对于并非每天都编写代码的人。更深层次地说，软件是某些事物的模型或象征，像画作一样。对画家来说，技术卓越也许就是关于他对所用画材的理解，以及何时去使用各种技巧，但是除此之外，更重要的是需要考虑其作品的意图。

软件也是如此。在软件中，技术卓越意味着许多依靠多年学习和专注才能掌握的东西。但是时间和专注不能确保一个人成为出色的开发者。

本书第二部分讲述了一些最为精髓的实践以及为什么它们如此重要。我们的重点是通过理解这些实践的原理以确保它们物尽其用。在本书的最后你会明白如何用这9个实践成功地构建软件。

① http://www.infoq.com/news/2012/04/Agile-Resources-Microsoft

3.7 总结

聪明人，新想法，我们首次感受到这个日益壮大的社区，这个社区由那些意识到问题并且想办法解决问题的人组成。但是这就是我们所希望的“银弹”吗？或者是我们干脆放弃寻求指路明灯的想法？

本章中心思想如下。

- 我们面临着巨大的挑战，一些聪明的人已经想出了一些办法开始帮助软件产业走向正确的方向。
- 敏捷方法论提供了传统瀑布开发模型的替代模型，用迭代方法构建软件可以降低开发成本。
- 编写代码需要客观性的技能，软件开发需要主观上的艺术创造性，软件开发者需要在这两者之间保持平衡。
- 尽管过去了十五年，敏捷依然在跨越从创新到主流的“鸿沟”。
- 软件开发者和管理者需要追求技术卓越，并以创造优质软件为**目标**。

敏捷软件开发直接面对瀑布模型这样的重流程方法论的挑战，通过提供一个基于技术实践轻量级过程，以生产出可维护的代码。但是许多敏捷团队没有重视这些技术实践，或者误用了它们，导致最终没有得到预期的收益。我们必须明白这些技术实践背后的原则才能正确使用它们。

第二部分

延续软件生命（和价值）的 9 种实践方法

如何才能将一件新的事物变成一个通用的实践？如何才能不仅仅学习新的实践而且从中获得最大收益？如何才能将这些新的想法融汇贯通变成优秀的习惯？

有些开发者比其他的人更高效，我花了生命中大部分的时间用来探索是什么让这些卓越开发者如此出色。我发现这些人并非生来如此，他们仅仅是和我们其他人有着些许的不同。如果我们理解了他们所理解的，学习他们的原则和实践，那么我们也可以达到类似的卓越成效。

第 4 章

9个实践

软件构建相当复杂，也许是人类接触过的最复杂的活动。编写软件是一项需要大量技能和练习才能成功的学科。

对软件的理解很容易出错，因为虚拟世界与物理世界差别太大了。在物理世界中，我们可以轻易理解构建一样东西的所需，但是在虚拟世界中同样的事情可能会变得非常困难。软件开发这一专业刚刚开始摸到门路，正如几百年前的医学一样。

不到两百年前，医学界还在嘲笑伊格纳茨·塞麦尔维斯关于微生物导致疾病的理论。外科医生术前洗手这样的小事怎么可能会关乎病人的生死？

医学界持有这样的观点，部分是由于细菌学尚未出现，同样也是因为当时的医学界在努力消除疾病是由于那些看不见的恶灵导致的传说（真相和传说经常有很多共同点）。所以，术前洗手并非外科手术的必要程序。

美国内战时期的战地医生了解细菌学，但是他们辩称没有时间给器械进行消毒。如果一个士兵急需截肢，他们没时间清洗锯子。但是当医学界研究内战时期战地手术的成功率时发现，死于感染的士兵多于死在战场的，于是医学界必须重新权衡消毒的重要性。

掌握了细菌学后，我们就明白了**为什么**必须清洗**所有的**器械。这是遵循实践（消毒某一特殊器械的**行为**）和遵循原则（对所有器械进行消毒的**理由**）的差别。

遵循软件开发实践如同消毒器械一样，如果想要事情顺利，就必须把**所有的**实践执行好。如果我们忽视了其中的一项，后果都可能不堪设想。一个细菌可能导致病人死亡，一个bug也足以让程序崩溃。这就是我们需要纪律的原因。

我并不相信开发软件只有一种正确的方式，正如也不仅仅有一种方式可以医治病人、进行艺术创作或者建造桥梁。我认为所有的事情都不只有“唯一方式”，尤其是编程。

尽管如此，在和数千名开发者共事之后，我亲身了解到我们是如何一次又一次重复发明轮子的。软件开发吸引了大量不同背景的人们，这给创建软件带来了大量不同的视角。同时，开发者之间的大量个体差异也会成为问题。企业级软件开发需要大量对于细节的关注以及与这些细节相

关的大量协调工作。我们必须对问题有着同样的理解、同样的实践方式以及相同的术语。我们必须有着相同的目标，并明确我们重视质量和可维护性。

有些开发者比其他的人更高效，我花了生命中大部分的时间来探索是什么让这些卓越开发者如此出色。如果我们理解了他们所理解的，学习他们的原则和实践，那么我们也可以达到类似的卓越成效。

但是从何入手?

软件设计是多层次而且复杂的主题，想要设计得当需要汗牛充栋的背景知识。进一步说，一些关键概念尚未被所有的开发者所熟知，我们中的许多人还在软件开发的门外摸索。

从很多方面来说，遗留代码的产生是因为我们有着这样的概念：代码的质量并不重要，重要的是软件正常工作。

但这是个错误的认知。如果软件会被使用，那么它便会被修改，所以它必须被编写得可以被修改。大部分软件并不这样。许多代码纠缠不清，所以无法独立部署或扩展，维护起来成本很高。虽然正常工作，但是由于其编写的方式导致难以修改，所以人们只好蛮力修改，让它以后的修改成本变得更高。

我们想要降低软件持有者的开销。根据南加利福尼亚大学的Barry Boehm所说[①]，交付之后bug的修复成本是需求设计阶段的100倍。我们必须找到一些方式，通过将代码变得容易使用来降低软件支持的成本。如果想降低软件持有者的开销，我们必须关注软件的构建过程。

4.1 专家知道些什么

专家们用特别的方式整理知识。他们通常用自己的词汇来彰显他们的特别之处。他们喜欢使用比喻和象征，通过他们的经验总结出核心信念。他们的**认知上下文**和我们普通人不同。

所有技术专家运用的技能都是可以被学习的——当你明白了专家行为背后的依据时，你可能会得到同样的结果。

专家级软件开发者，那些不仅仅是比常人优秀一点而是获得巨大成果的人们，他们对软件开发的看法与我们不同。他们关注技术实践和代码质量，明白什么是重要的、什么是不重要的。

最重要的是，专家级软件开发者对自身的要求比我们严格。

我惊奇地发现，我认识的最优秀的开发者也是最整洁的程序员。我原以为编码速度快的人肯定粗心大意，但是事实却相反。我遇到过的最快速的程序员特别注意让他们的代码保持容易维护。他们不仅仅是在类的开头声明**实例变量**，还会按字母表顺序（或者其他便于理解的方式）排列

① Boehm, Barry, and Basili, Victor R. "Software Defect Reduction Top 10 List." *Computer*, Vol. 34, Issue 1, January 2001. https://www.cs.umd.edu/projects/SoftEng/ESEG/papers/82.78.pdf

变量，频繁地将方法重命名而且反复调整其位置直到找到最合适的地方，并且将无用的代码立即删除。

即使发觉了这些编码速度快的人的代码整洁程度也很高，我也是在一段时间之后才意识到这两者之间的因果关系的。他们不会**不顾**代码质量而加快速度，反之，**因为**他们保持了代码的高质量才能保持快速编码。意识到这点会影响我们对软件开发的认知。

大多数人都知道用一个经过深思熟虑的方式解决问题会得到长期的收效。然而他们不知道的是，收效来得远比预期要快。在现实世界中，我们用愿意为其花费多少来衡量质量。高质量的物品会持续更长时间进而更加昂贵。但是虚拟世界则不同。

在虚拟世界中，对于质量的关注通常会节约长期开销，同时也会节省短期花费。这并不是说开发者不能时不时地做出些短期的妥协，而是当他们做出妥协时要清楚地知道，每次他们回头面对糟糕的代码的时候要付出的代价。如果代价过高，他们也许就会想要在进一步增强软件之前回去清理代码。

商业中会权衡成本收益，软件也应如此。和其他资产一样，软件也应该经常维护，以免成为累赘。

4.2　守–破–离

精湛的技艺并不仅仅是技巧和能力。日本格斗技“合气道”将精通的三个阶段定义为：**守**、**破**、**离**。

“守”是基本架势，基础知识。电影《龙威小子》中的“上蜡，刮蜡”就是“守”阶段的例子。电影中的年轻学徒丹尼尔被要求反复地给车打蜡。他没有被告知，这样的练习是在为达成最终目标做准备。直到他掌握了架势之后才知道原因。

人们常常通过一系列“应该这样做，别那样做”的规则学习敏捷。那仅仅是学习的第一阶段，但是很多人认为只要学了一些规则便可以进行敏捷了。

像软件开发这样的复杂活动很难用一些规则定义。软件开发中有许多禁忌，一种情况下好用的方法在另一种情况下可能很糟糕。所以，软件开发者的成长之路通常对应着一条很长的学习曲线。

人们从“守”开始的原因是，这些实践背后的理论并不是那么显而易见。在格斗技中，你需要的不仅仅是明白理论，还必须把理论付诸实践。在合气道中，这被称为“破”。软件开发也是如此。想要成功，我们必须理解理论——实践背后的原则，这样才能让实践变得有意义。

不能把“破”当作一组规则教条地学习。它必须从经验中汲取，同样也能从他人的经验中习得。

一旦使用了这些实践，深刻理解其中蕴含的理论，这些实践和理论就会相互融合并达到合气

道的精通境界：离。只有通过不间断的学习才能达到真正的大师级境界。

Malcolm Gladwell在《异类》[Gla08]中提出，在智力密集领域内需要超过一万小时的练习才能熟练掌握。我们不需要再思考，因为它几乎成了我们的第二本能。如果任何的复杂活动都需要一万小时才能精通的话，那么软件开发也不例外。

毕加索也知道这个道理。只有在学习了绘画的规则之后才能打破规则。毕加索创造了前所未有的画作，但是他一开始接受传统画师训练的事实却鲜为人知。他可以用传统大师的方式绘画，并且终其一生遵循这些传统技巧。但是他并未满足。他将这些技巧升华至另外一个高度。想要打破规则并开拓新领域，必须精通现有的规则。

软件开发也是如此。软件构建中有许许多多的规则限制和技巧。和其他的人类创造一样，计算机程序是某些事物的模型。我们对**物理**模型习以为常，但程序同样也可以是**行为**模型。

为了准确建模，我们首先必须了解建模的目标，同时需要理解我们有哪些建模技能或者方法。我发现把这些方法分为两类即原则和实践，比较便于理解。

4.3　首要原则

首要原则最早是由马可・奥勒留在讨论“黄金法则”的时候提出的：“对待别人就像你希望别人对待你一样。”黄金法则之所以是首要原则，是因为我们的法律、我们的社会甚至我们的文化都构建在这短短一句之上。其他的原则都可以通过首要原则引申出来。

黄金法则是我们法律的一个大前提，是追求正义的基础。想象一下，如果没有了黄金法则，人人都只顾一己私欲，法律会变成什么样。理解并且认同这个法则是所有学科成功的核心。

目前尚未出现软件开发的黄金法则或者希波克拉底誓言，我们依然在试图区分主次，区分哪些事情应该注意，哪些可以安全地无视。对于一个年轻而又全新的领域，这些要求似乎有些高。但是我们已经开始建立起一些关于软件开发的基本原则。

软件开发首要原则的例子是**单一职责原则**[①]：“修改某个类的原因有且只有一个。”

这看起来虽然是很简单的一句，但里面饱含深意。因为类作为系统中对象的模板，意味着我们需要把它设计成代表单一**事物**。这意味着很多事情。这意味着我们的系统中会有许多小巧的类，每一个类都专注于实现单一的职责。

通过将类的职责变得单一，我们限制了这个职责和系统中其他类的交互。这让这个类变得更容易测试，更容易找到 bug，而且便于将来的扩展。“单一职责原则”引导开发者设计出分离性好的、模块化的系统。

软件开发的另一个首要原则的例子是Bertrand Meyer在《面向对象的软件构建》[Mey97]中表

① http://www.butunclebob.com/ArticleS.UncleBob.PrinciplesOfOod

述的**开闭原则**："软件实体（类、模块、函数等）应该对扩展开放，对修改关闭。"

这意味着设计的系统应该允许在不修改原有代码的前提下轻易扩展。当我询问开发者为什么"开闭原则"很重要时，他们马上就知道了原因：因为修改现有代码往往很困难，而且比编写新代码容易出错。当开发者理解并且重视"开闭原则"的时候，他们倾向于编写出更容易维护的代码以降低日后扩展的成本。

原则非常强大但不易于实施。原则告诉你做什么但没告诉你怎么做。在软件中，有很多方式实现"开闭"。它将推动我们构建高内聚对象，进行抽象编程，保持行为之间解耦。值得注意的是，这些特性都是我们为了实现"开闭原则"过程中的副作用。

同样，也有许多实现"单一职责原则"的方式。遵循这一原则会让开发者从问题领域内发掘出更多实体、隔离行为、使系统更加模块化。所有这些都有助于构建更具弹性的系统。

原则通常不言而喻。开发者也许对他们所要达成的目的有着模糊的认识，但是通常在头脑中没有清晰的概念。可以把这些原则当作指导我们做正确事情的大智慧。

4.4　关于原则

原则可能是清晰明了的，也可能是模糊不清的。无论表达的方式是否优雅，原则给我们指明了正确的方向并且让我们距原则所追求的本质更进一步。所谓的"我们"不仅仅是软件开发者，而是软件开发团队中的每个人以及在他们周围和他们有联系的所有人。

原则帮助我们把某件事情通用化，有助于梳理知识体系。并不是所有的原则都是对等的。有些更加纯粹，比其他的更基础。正如我们前面所看到的，可以引出其他原则的原则就是**首要原则**。

我认为原则是些高层次的目标。因为知道原则的价值，所以我们要努力实现。我们也知道，虽然原则是有价值的目标，但并不总能实现。在软件中，原则代表着帮助开发者构建更好软件的指导性建议。

4.5　关于实践

原则很重要，但是光有原则是远远不够的。我们同样需要在实际场景下达成原则的方式，这就是实践的作用。

我用严格的标准来定义实践。若要成为一个实践必须：

- 在多数情况下产生价值；
- 容易学习且容易传授；
- 简单易行——简单到**无需思考**。

当一个实践符合这三个条件时，它就很容易在团队中推广并且使团队受益。只需要**使用**某个

实践，然后自然而然就能够在日常工作中节约时间和精力。

本书中的9个实践代表了一组极具价值的实践，它们经常被误解和误用，但对可持续的生产至关重要。每个实践都可以节约大量时间。我并不是要给开发者增加更多的工作。他们已经应接不暇了。这些我建议开发者采用的实践，无论是短期还是长期都可以节约开发者的时间。它们帮助开发者构建清晰的、可测试的行为。

4.6 原则指导实践

开发软件是一项面对着无数个问题和抉择的工作。质疑是一个有效但是令人疲惫的过程。我应该做这个任务还是另一个？所以评估十分重要，可以让我们获得新的想法并得以创新。

但是不停地自我质疑是很累人的。我并不主张开发者完全不假思索，但是如果有一些通用的实践可以快速地不加权衡地实施，那将会十分有帮助，而且会让你距离首要原则更进一步。举例来说，消除冗余这个简单的实践可以帮助统一定义出单一职责的类，让你更加容易实现“单一职责原则”。我训练自己，为的是让自己迅速察觉并消除冗余代码。我不需要问自己是否要消除它们，因为这么做已经成为了习惯，这个习惯让我的工作更有效率。

当你理解其背后的意图之后，所有我推荐的实践都可以无需多加思考地实行。这会给你规划和构建软件带来更好的帮助。实践用行动取代了质疑和不确定。

原则可以指导我们如何把实践的效果最大化，帮助我们将实践物尽其用。原则就像指路明灯，给我们展示了实践的正确使用方式。

投资的基本原则的一个例子是“低价买，高价卖”。这确实是个好的投资建议，但也是个蹩脚的建议，因为它没说明**如何**才能做到。

原则给了我们目标，而实践指导我们如何达成目标。我们可以实际**实施**实践，投资实践的一个例子是**成本平均法**。如果每个工资周期，即每个月——一个固定的时间周期——我把收入的一个固定比例进行投资，当价格走低的时候我则用一样的投入买入更多的股票。假设股票市场持续攀升，我在大多数的情况下都实现了“低价买”。如果我一直持有这些投资直到准备退休，并且这些股票的平均价格高于我这些年来购买时的价格，我实现了“高价卖”。原则（低价买，高价卖）和实践（成本平均法）起到了相辅相成的作用。

实践帮助我们距原则更进一步，原则帮助我们正确实施实践。如果不同时注重这两者，我们就容易误入歧途。要么成为不切实际的理论家，要么成为没有头绪的实干家。我们必须保持二者之间的平衡。

本书的每个实践都有着重要的目的。当你理解了实践背后的目的时，你就会知道如何正确使用它们，什么时候应该使用，又或者有哪些其他选择。实践是原则的应用，它们基本上算是原则的化身。

4.7　未雨绸缪还是随机应变

如果没有一组可以用来创建可修改代码的实践，我们就无法轻松地修改代码，当必须修改的时候需要付出很大的代价。想要日后方便修改，必须在修改发生前做出预计。这给人很大压力。

当一个团队在开发一个很大的功能的时候，想着“好吧，全都通过编译了，天哪，我希望别出错。真希望知道最后会变成什么样子”，会有很大的压力。

压力对于构建更好的产品毫无帮助。整个软件开发产业都建立在对变化的预估上，而不是用一组经过时间检验的原则和实践来更好地应对变化，当开发者意识到这点的时候，就明白了这个年轻的产业还有很长的路要走。

这是非此即彼的：未雨绸缪还是随机应变，大多数开发者都不知道如何让软件更加容易修改。想象一下，如果要求每个演员都是一条就过，那么他们拍摄电影时将会承受地狱般的压力。仅仅知道你不需要“一次就完美”而是“只需合格即可”，就会让事情简单很多。通过消除（或至少是大量减少）由于担心表现所产生的压力，开发者基本上可以一次就做好，或者发现新的想法，或者发现新的做事方式，并且敢于接受更多的挑战，因为现在他们知道了即使一开始失败他们依然能够挽回。

我们多数人并不能预知未来。在团队交付一开始要求的内容之后，用户想要什么谁也不知道。预计将来的需求令人疲惫，而且多数情况下你都会搞错。但是我也发现很多的开发者试图为他们的代码预计将来的需求，即使他们的预计并不是当前所需要做的。这可能导致开发者将时间浪费在担心一些他们目前并不需要的功能上，并且占据他们本来应该花在当前迫切需要做的事情上的宝贵时间。

如果开发者能够停止揣测用户将来可能要什么，找到方法让代码在需要的时候可以变更，会怎样？如果有一系列的原则和实践让开发者可以遵从，甚至不需思考，是否能让代码修改变得容易些？那么当客户无法避免地想要新功能的时候，代码便可以满足新的需求。

这不仅仅是想当然。我相信开发者必须而且可以有一系列通用标准和实践帮助他们应对变化。我们在本书后面将会了解到很多这样的实践，并且通过这些知识，你可以自己发掘出更多这样的实践。但是在开始深入这些实践之前，我们必须对软件中“好”的定义达成一致，以便理解其原则——使用这些实践的理由。

4.8　定义软件中的“好”

是什么让软件变“好”？当开发者看到一种设计或一段代码的时候，他们如何判断写得是否优秀？他们关注的是哪些东西？

当我询问开发者这些问题的时候，我很少得到一致的回答。对于一些人来说，“好代码”意

味着速度快、效率高。对于另一些人来说，“好代码”意味着容易阅读并理解。还有些人说“好代码”是没有bug的。

这些都是好代码，但是如何写出好代码呢？在必须做出取舍的时候，底线在哪儿？这些是不常问到而且很难回答的问题，但是它们影响着管理者和软件开发者的日常工作。

软件的外部质量诸如用户体验——可用性、少有异常、更新及时等——是软件内部质量的表现。即使用户并不直接体验软件的内部质量，他们也会受到影响。内部质量低下的软件也让开发者难以维护。

早在20世纪80年代，我使用一款叫做Clipper的dBase III 编译器[①]。那是一款非常成功的产品，让Nantucket软件公司赚取了数千万美元。但是无论开发者多么优秀，由于代码的复杂性他们无法进行重大升级。最终他们失去了市场机遇，然后关门大吉了。

很长时间内，开发者都以为软件不需要修改，觉得开发软件是一件**一蹴而就**的事情。但事实上如果软件有人使用，就很可能需要修改。这是件好事，它意味着用户找到了从软件中获取更多价值的方式。开发者希望通过让代码容易修改来满足用户。

人们觉得有价值并且经常使用的软件很可能需要在将来进行修改。正因为如此，开发者必须关注增强内部代码质量的标准和实践，这会让软件更容易维护。当我问到开发者他们在写代码的时候希望达到什么样的内部质量时，我得不到统一的答案。这是一个年轻的产业，而且尚未得出所有人都认可的开发标准。

我们必须对什么是“好”达成共识，并不是仅仅针对个人，而是针对整个产业。达成共识之后，才能对软件开发的基础目标达成一致。

鉴于在已有软件中修改代码和添加功能的开销之高，可以说，让好的软件按照人们期望的方式运作**并且**可以被修改，以便从容面对未来的需求，高于一切。让软件变得可修改，能提高创建软件时的投资收益率（ROI）。

软件是一种资产，而资产的价值不仅仅取决于我们现在能从中获取的价值，也取决于将来能从中获取的价值。“生命周期管理”在房屋建筑学和产品设计学中扮演着十分重要的角色，所以它在软件设计学中的地位也至关重要并不足为奇。

当然，软件开发者经常无法预知他们代码中的哪些部分将来会需要修改。因此，他们需要把**所有**的代码都写成可修改的。开发者要达到可变化性，需要理解什么是高质量软件，软件如何随时间变化而变化，还要从相应的原则和实践中培养习惯。

许多人通过外部标准衡量软件质量，诸如正确执行、无bug、速度快等。但是这些都是一些深层次原因的表象。我们在本书中讨论的软件质量是**内部质量**，通过提高软件的内部质量可以更

① http://en.m.wikipedia.org/wiki/Clipper_(programming_language)

加接近我们当作外部质量来衡量的东西——内部质量才是原因而非结果，它们潜藏在优秀的软件之下。

内部质量有时候很微不足道，但是加在一起构成了优秀的软件开发原则与实践的核心。仅仅说句你们“敏捷了”远远不够。敏捷方法论包括了管理实践，但是极限编程中的技术实践才是敏捷的核心。为了真正看到这些聪明的新想法的成效，我们必须理解敏捷实践背后的原则。

本书中，这些原则，也就是为什么这9个实践会改变开发软件方式的原因听起来都非常显而易见，它们都像是“低价买，高价卖”。这9个实践会帮助你构建无bug的软件，更容易低成本维护和扩展：写得好、风险少。

4.9　为什么是 9 个实践

Scrum提供了一个支撑敏捷开发的最小框架，而极限编程一开始就提出了12个核心实践。

我进一步把它们浓缩成了9个最佳实践。头两个是多数人在想到Scrum的时候想到的，剩下的7个是技术实践。这9个实践是为了帮助我们思考正确的软件开发方式而设计的，从而缩短发布周期，这是对传统观念的颠覆。

9个实践如下：

(1) 在问如何做之前先问做什么、为什么做、给谁做
(2) 小批次构建
(3) 持续集成
(4) 协作
(5) 编写整洁的代码
(6) 测试先行
(7) 用测试描述行为
(8) 最后实现设计
(9) 重构遗留代码

有人说，人们能够将他们的注意力同时集中在差不多7（上下浮动2）件事情上，所以9件事情大概是多数人可以记住的极限。9件事情也同样不多不少是我这本书能够容纳的。这9个实践是我发现的价值最高但最经常被误解和误用的。

你并不需要实施全部9个实践，但必须理解实践目标。如果有其他方法避免其中某一实践解决的问题，便可以安全地替换这个实践。但是和毕加索一样，在打破规则之前必须要理解他们。

你可能会发现这9个实践的一个通用主题，那就是构建可验证的代码行为。这有助于将开发的注意力放在做正确的事情上，与此同时容易验证并且便于在将来修改。

这9个实践有助于在保证软件开发流程顺利进行的同时创建可修改代码。当然也有许多其他的实践有助于编写可修改的代码，但这9个提供了一个坚实的基础，为降低构建和维护软件的成本开了一个好头。

本书的目的是帮助你更有效地对软件构建进行思考，这样你就能提高软件开发效率，并且让软件便于日后扩展。

4.10 总结

本章着眼于优秀的软件开发过程的核心。它们帮助人们把开发注意力放在编写可维护的软件上面。

本章中心思想如下。

- 如果软件有人用，那么它便需要被修改；所以必须编写出可修改的软件。
- 为了要对某些事物准确建模，我们必须先理解它。
- 成为一个出色开发者需要的所有条件都是可以学习的技能。
- 与其试图预计将来可能的修改，我们应该研究出一些工程实践帮助软件更好地应对修改。
- 软件开发面对着不同于其他学科的独有挑战，为了应对这些挑战，我们必须理解这些实践背后的原则。

本书将讨论的这9个实践直接对应着构建软件所面临的许多挑战。这些实践的关注点并不是让开发者的工作更加迅速，重点是构建出更容易维护和更容易扩展的代码。关注代码质量让代码更容易维护，从而可以让开发者的工作更加迅速，不仅仅是在短期内，而且是在他们编写软件的整个生命周期中。这会减少他们编写的软件演变成遗留代码的可能性。

第 5 章

实践1：在问如何做之前先问做什么、为什么做、给谁做

传统的软件项目中，高达50%的开发时间花在了需求收集上，这让开发从一开始就陷入了颓势。当团队在专注于需求收集的时候，他们把经验最少的人——对编码没有技术背景的业务分析员——派出去和客户沟通。他们不可避免地跟着客户鹦鹉学舌。

我们都有着跟别人讲他们喜欢听的话的倾向，有些面对客户的专业人员依然会陷入这个陷阱。像这样跟客户沟通会让人觉得不舒服——“不，你并不想要那个，你想要的是这个”，或者，“别担心具体怎么实现，交给我们的开发者就好了”。

把客户要求的功能列出来并且告诉客户他们想得到的答复“知道了。那个我们能做”会更简单。但是我们真能做到吗？

更重要的是，我们**应该**对客户言听计从吗？

在实际情况中，我们习惯用具体实现的字眼沟通。这是人们说话的习惯，而且很难发觉。我从我特定的理解出发给出了一个总结。你听到那个总结然后用你特定的方式理解。最后我们从两个不同的主观理解出发以为有了共识。但是它们可能完完全全不同。

需求没有办法验证，而且是经客户讲给分析师，分析师把它写下来，然后开发者阅读需求后再转变成代码。这简直是个传声游戏。对于同一事物的看法千变万化，所以当你最终将交付版本交给客户的时候，他们可能会说：“我不是那么说的。那不是我想要的。”

5.1 不要说如何

客户与客户服务经理都不应该指挥开发者**如何**做某件事，理由如下。

对于不了解软件的人，软件开发的过程是完全不符合直觉的。并不是随便什么人都能坐到电脑前把软件搞清楚。大量的精力花在了将软件开发流程用通俗的方式讲给外行人上，这被作为一个手段让客户感受到他们的需求被听取，并且让项目是什么以及到底做什么有着明确的共识。这

样做理由很充分，但不幸的是，这样往往产生僵化的需求。

需求听上去是个好主意，但是因为人们习惯于用**如何**做来交流，需求引发的问题往往比它解决的多。而且，问题是字面意义上的问题。这是人类自我认知的工作方式，而非软件需求独有的。

一旦开发团队听到或看到描述**如何**做事的需求，他们就被束缚了双手。这等于直接说“用这样的方式做”。然后开发者照着编码，通常情况下他们机械性地堆砌代码，而不会后退一步去问一句“我怎么能够创建一组交互的对象以实现这个行为呢”。

软件开发不再是告诉计算机去干这个干那个了，而是创建一个计算机根据一组相关对象的交互行为而执行的**世界**。

这听起来像电影《创：战纪》一样，我不想把软件拟人化——众所周知计算机没有意识——但是如果你创建了一个山坡对象和一个球对象，并且建模得当，这个球就会顺着山坡滚下去。

同样的道理也适用于业务规则。

业务规则是系统的规则，是每条`if`语句中包含的代码。业务规则告诉系统如何执行。

软件开发者依据建模的领域来表示这些规则。无论我们的软件针对什么领域或者业务进行建模，我们都愿意使用这些领域特定的术语来定义知识库。我们希望业务规则自然而然地被我们软件模型中的对象所表现出来，我将其称为**问题域**。

这应该不难理解。想象一下真实世界中的对象和物体。建筑师依照图纸建造，文员用文件夹整理文件，都是按照对他们有意义的方式做事。建筑师不会上下颠倒或者从反面看图纸或者画出比例不对的图纸，文员不会因为P代表了Person（个人）而把“Bernstein, David”的文件放到字母P下面，或者因为姓氏中有字母R就放到字母R下面。

软件开发者也是如此对待软件中的对象的。

虚拟程序世界中的对象应该对应着它们所代表的现实世界对象。

5.2　将“如何”变为“什么”

我们都应该追求一种和客户通力协作的合作关系。但问题依然出在表达方式上。人们很容易就陷入讨论如何做事的模式。而如何做事是软件开发者自己的事情。

作为软件开发者，我们希望了解产品负责人和客户需要**什么**以及**为什么**需要，而且我们想知道这些东西是为**谁**而做的——我们并不希望他们告诉我们**如何**去做，因为那是我们的工作。这就是软件开发者的世界。我们跨越**什么**和**如何**。只有我们知道**如何**去做。

如果我想要给自己建造一座房子，我会雇一个设计师、一个建筑承包商、一个管道工、一个电工，等等。我会告诉他们我希望厨房有很大的料理台和一个煤气灶，我希望有若干个卧室，一

个按摩浴缸，等等。

如果设计师问我屋顶的角度应该是多少，建筑承包商问我需要用到多少钉子，管道工问我需要订多少管道。我的回答会是“按照规矩来，做好就行”。如果设计师问我建筑规范是什么，我则会另换一个设计师。我不知道建筑规范究竟是什么，但是我愿意——并且很高兴——雇用一个经验丰富、深谙其道的专业人员。我们信任那些规范，即便不知道规范是什么，也能信任专业人员会把事情做好。

即使没有针对软件构建的“规范”，依然有很多经验丰富的专业人士不仅可以处理“如何”做，而且可以提供新的功能、新的想法和新的途径，以便让最后的成果比一开始预计的好很多。

软件开发者对实现和抽象同样精通。在开发过程中，开发者会产生许多对于非开发者来说毫无意义的取舍，因为优秀的开发者首要考虑的是可维护性。

而且做事情的方式远远不止一个。

几乎不可能让不同的团队构建出一模一样的软件。我可以给一千个不同的开发者一样的需求——我曾经这么做过——我会看到一千种不同的实现方式。结果可能看上去一样；事实上，几乎**肯定**一样，但是他们实现的方式是多种多样的，有时甚至千差万别。这是好事，更新、更好的想法就是如此产生的。但同样需要一些共同基础，一些为了软件能正常构建而在一开始设立的标准和实践，它们不过多地依赖具体实现。

到写作本书的时候，我已经在课堂上教授过8万多名开发者了，我的课程通常都会有一些计算机实训。目前为止，有差不多500人完成了我要求的编写一个在线拍卖系统的实训课程。这并不是一个轻松的任务。

我发现解决方案有着惊人的多样性。

他们中的多数解决方案会有着许多相同点，并且对实体的称呼基本一致——拍卖、拍卖者、竞价——但实现却大不相同。有些会保存一份登录顾客的列表；有些则跟踪所有的用户对象，无论登录与否。

这些事情没有所谓标准答案。我不关心某一位开发者是如何实现某一功能的，我关心的是所有开发者都能理解他们选择的方式和未选择的方式之间的利弊。

除了一组核心的原则、模式以及开发者应注意的实践之外，软件开发者并不需要把他们的实现“标准化”。但他们应该把如何定义行为以及编写什么测试进行“标准化”。如果开发者对于某一特定行为应该编写什么测试能够达成一致，那么他们就有了坚实的共同基础。

5.3　要有一个产品负责人

在我接触过的所有**大型**软件开发项目中，都会有产品负责人。这个角色有很多叫法：产品推

动者、客户代表、驻场客户或者项目经理，有时候甚至称为团队经理或者团队带头人。你怎么称呼这个人都行，我会按照Scrum中的惯例称其为产品负责人或者PO（Product Owner）：一个对整个项目全权负责的人，和客户接触**最频繁**，对产品理解最深刻的人。产品负责人是产品的权威人士，这个角色至关重要。

我曾在IBM工作过许多年并且开发过由委员会设计的项目和产品，我可以肯定地告诉你，委员会设计根本行不通。

毕竟，**大多数**由委员会设计出的东西都行不通。

产品负责人并不仅仅负责引导开发流程，还需要推动整个流程，产品负责人也**不**一定必须是技术人员。事实上，如果产品负责人并非技术人员，而是对产品本质有着深入理解，是**真正**熟悉相关领域知识的人，那样会更好。

产品负责人还要准备好承受压力。

一方面，产品负责人是一个超级明星。另一方面，产品负责人对产品来说生死攸关。有时候产品负责人被称为产品的**七寸**。这个角色必须有绝对的权威。即使产品负责人有时会出错，软件开发者也需要得到关于问题明确且直接的回答。开发者身处细节的迷雾之中，多数人并不思索细节，更别说真正理解。

产品负责人是沟通的中枢。所有人都向他更新近况、提出问题，他负责过滤这些信息。产品负责人控制产品的版本，定义下一步构建什么，尽管定义**产品本身**需要和整个团队协同完成，但必然是由产品负责人和其他利益相关的特定领域专家（subject matter expert，SME）推动的。

产品负责人会说："这是下个阶段最主要的功能。"

产品负责人排定待处理任务和需要构建的功能的优先级，保证重要的事情优先处理，不重要的事情放到后面。有序的待处理任务列表的好处不仅仅是让重要的事情优先，而且保证不重要的事情靠边而不浪费我们的时间。

这部分是我们这个产业尚未充分揭示的地方。

在一个电影团队中大家都各司其职，每个人都是不可或缺的。但是想要创作一个好的故事、一部真正伟大的作品，则需要了解故事走向节奏的导演赋予电影生命。产品负责人就像电影导演一样，是对整个项目负责的人。

开发者善于提出没人想到过的问题。为了让编写的软件正常工作，他们必须事无巨细地把这些问题提出来。如果他们没有这样做，没能考虑到一个可能的条件或潜在的问题，就没法保证计算机会正确运行，通常会导致程序崩溃。

有时候开发者会把事情搞错，不是代码出错，而是**代码应该做的事情**没搞清楚，这时就需要产品负责人回答所有问题，并且把控正确的方向，即使很多时候都没有绝对正确的答案。

我们需要引申用户的意图，进而不要沉浸在细枝末节或者具体事务中。

话虽如此，但是在没有标准指导下工作还是会让开发者难以接受。让开发者在拿到全面需求、**知道**要做什么之前就开始编写软件，听上去很低效甚至很唐突。但是开发者可以在需求并未完备的时候就开始工作，一边开发一边沟通，这么做效率很高，而且会明显地提高产品质量。

这就回到了要问“做什么”而不是“如何做”的问题上。

既然如此，那么需求文档的替代品是什么？

5.4　故事描述了做什么、为什么做、给谁做

故事是包含如下内容的一句话：

- 某个东西是什么
- 为什么会有某个东西
- 这个东西是给谁做的

假设我们要创建一个线上电影售票系统。故事应该是：

> 作为一名电影观众，我希望能在线购买电影票而不用在电影院排队。

这一句话告诉我这个软件是为谁而编写的（电影观众/消费者），他们的要求是什么（在线购买电影票），他们为什么这么要求（避免在售票处排长队）。这是个良好的开端，但仅仅是开端而已：并没有足够的信息用来开始编码。所以故事并不能取代需求文档，而是关注需求的上下文：**做什么**，而不是**如何做**。

编写出高质量故事的方式有很多，本书就不再赘述了。一本关于编写用户故事的好书是Mike Cohn的《用户故事与敏捷方法》[Coh04]。

Alistair Cockburn说过用户故事是“沟通的保障”①。我们并没有足够的信息用来构建功能，但我们有足够信息用来开展针对功能的沟通。

需求文档并非被用户故事取代，而是被产品负责人和开发者之间的交互取代，被产品负责人和客户之间的交互取代。这些交互是构建软件所需的充分理解的来源。

顺便提一句：故事通常写在一些3 × 5的卡片上，所以没法写很多内容。我常常也给团队配更大的书写笔，强迫他们把字写大些。我们不需要那么多细节，所以提倡想办法让添加“如何实现”这样的细节变得更加困难。

将一切流程都长篇累牍地用文字书写下来，但等到它被编码出来之后就变得不准确了。故事

① Cockburn, Alistair. “A user story is the title of one scenario whereas a use case is thecontents of multiple scenarios.” http://alistair.cockburn.us/A+user+story+is+the+title+of+one+scenario+whereas+a+use+case+is+the+contents+of+multiple+scenarios/v/slim

并非构建一组开发者并不需要而且最好不要的指令，而是能够让我们把精力放在工作上，所以开发者是用一种探索的方式编程，这种方式令人振奋。

振奋而且高效。

传统的瀑布模型开发过程中，计划本身需要花费时间，所以在计划阶段完成后会有明显的完成感。创建需求的相关人员觉得他们的任务完成了，但是他们所做的仅仅是让人们**以为**对于项目完成之后应该是什么样的达成了共识。

之后开发者坐下来开始编码，意识到需要回答的问题中只有25%得到了答案：我们是否想到了这个或那个——我们又该如何处理那个？

用户故事保证我们不仅仅在计划阶段，而且在**整个开发阶段**都保持对需求的关注。

正如在敏捷中所谓的“刚好够用的文档”。

许多团队花费了大量时间在需求文档和设计文档上，忘记了软件开发实际上全在于编码。这些团队最后让那些不连贯的文档和代码分离。实际上，代码本身应该**承载**着那些知识。他们过分关注用那些产出物来描述系统，而没有时间把代码本身变得通俗易懂又便于维护。

是沟通让开发过程运转，产品负责人和开发者通力协作的方式，其效率远远高于其他的方式。

用户故事是有限的，只代表针对特定用户的单一需求的单一功能，而需求文档可以很发散。用户故事是有限的意味着它也是可测试的，而可测试意味着你知道什么时候算完成。这对于软件开发者是十分关键的，因为如果我们不知道一个功能会以何种方式使用，就会过度开发。这就是现有软件中有将近一半的功能从未被使用的原因。

软件开发者之所以会过度开发，是因为我们**害怕**某个功能会以我们没有预料的方式使用。如果我知道了软件到底如何被使用，就可以正好针对这一需求开发，然后进行其他工作。

许多“如果万一”在软件开发过程中出现。这种不确定性给我们的产业带来了悲伤和不愉快。每次你需要**预期**的时候都是痛苦的开始，因为你永远不知道是否做过头了。你永远不知道是否做得足够。你永远不会得到相应的反馈。

难怪开发者老得比较快。

所以“把开发过程变成一个探索式的过程”是一件非常令人振奋的事情，而且是构建软件的强有力的方式。自然而然地，开发者开始构建原型，将开发成果逐步展现给客户，从客户中得到启发，得到各种各样的关于功能和性能的新想法。

软件开发者依然希望标准化，我并非影射启发式的过程是缓慢迂回的过程。我们可以并且应该构建验收测试——有许多自动化工具来替我们做——并当工具说“可以，通过”就万事大吉！我们就可以继续前行了。

我们不需要继续纠结“我是否覆盖了这样那样的情景”。

5.5 为验收测试设立明确标准

依照**刚好够用的文档**，团队在开始构建功能之前还需要几件东西。产品负责人需要了解以下内容，而非依赖按部就班的需求文档：

- 验收的标准是什么？
- 和开发者展开对话需要了解多少细节？

开发者对于他们要构建的软件需要了解相当多的东西。

产品负责人和开发者直接沟通的过程是无法自动化的，但我们可以把验收标准自动化：

- 功能是做什么的？
- 功能什么时候工作？
- 什么时候可以继续前进？

这让开发者进而关注边界情形——当输入极端参数时会出现的问题和情况。

客户通常很奇怪有人问他们这类问题。即使在之前的工作中遇到过这类情况，他们在构建需求阶段也从未想过这类问题。但开发者必须把这些事情构建到软件当中，以保证软件在**任何时候**都能正常工作，并不仅仅是当一切正常的时候——我们所谓的**快乐路径**。

快乐路径假设什么事情都不可能出错，但开发者必然会需要对付次要路径、错误路径、异常路径。一个功能中只有一个或者几个快乐路径，但可能会有很多的异常路径，所以用系统性的方式处理这些异常对于简化软件非常有帮助。

边缘情况是快乐路径的分支。作为一个开发者，也许我得到了超过界限的输入，也许我要连接的在线服务暂时不可用而导致错误。我该如何处理这些事情？我该如何反应？

当然，我不希望计算机宕机。我希望做些更有意义的事情。也许我把消息展现给用户，或者尝试另外的路径。开发者需要把各种支线过程填满，非快乐路径、错误情况，等等。这意味着我们要问：“**用户故事中有哪些部分会出错？**”

将这些分支路径放在你的验收测试中。你必须定义所有的边界情况以完成用户故事。当你犯了错误，或者没有处理边界情况，计算机就会崩溃。我习惯称之为“死之蓝屏”。

无论你管它叫什么，那并不是一番美丽的景象。

5.6 自动化验收标准

通过验收测试来描述需求，验收测试就容纳了丰富的信息。它帮助开发者理解用户真实的业务需求，所以我们能够正确编写代码，而且代码具有非常好的可读性。验收测试帮助定义系统行为，提供输入和期望输出的真实样例。我们不仅仅是在讨论理论。现在我们可以说：“给定特定

输入，应该得到特定输出。”这是切实可行的。

把抽象的东西具象化对软件开发来说是一个重要的技能；反之，把具体的东西抽象化同样重要。用验收测试的方式构筑出行为和实现让开发者可进可退。通常两个——最多三个——行为实例就可以开始编码了，足够我们理解和概括。使用自动化验收测试对软件行为进行标准化让每个人的理解都一致。产品负责人和开发者有着同样的预期。

无论你是否使用自动化测试，把验收标准和边界情况写在用户故事卡片上都是好的办法，可以用来提醒你需要处理哪些异常。知道一个功能在满足了特定验收标准之后就算完成了，这样可以让开发过程更加专注。开发者有时候倾向于过度开发，或者我们所谓的“镀金”，因为我们不清楚软件会如何使用或者是否足够健壮。同样，定义良好的验收标准可以消除这些问题。

所有的项目都需要一个超级明星产品负责人才能成功，就如同所有成功的电影都有一个优秀的导演。自动化验收标准消除了争论，帮助产品负责人和开发者把注意力放在对项目成功最重要的一些事情上。

5.7　让我们付诸实践

以下是把这些想法付诸实践的方式。

5.7.1　产品负责人的 7 个策略

每一个伟大的产品背后都有一个优秀的产品负责人。产品负责人控制着产品的远景和工作的优先级。有7个策略能帮助产品负责人更高效地工作。

成为特定领域专家

产品负责人必须是特定领域专家，对产品做什么要有深刻理解。产品负责人必须花时间构想整个系统，在开始构建之前通过实例来帮助他们理解。

在开发过程中探索

因为产品负责人必须控制产品的远景，他们在构建过程中必须对探索更好的解决方案保持开放的心态。交互式的开发过程提供了很多反馈的机会，产品负责人需要把握这些机会，把正在开发的功能构建到最终用户的手上，保证开发过程的正确方向。

帮助开发者理解为什么和为了谁

理解为什么会有一个功能的需求，以及这个功能是给谁做的，能帮助开发者理解需求的上下文。开发者通常会想出更好、更容易维护的实现，完成同样工作的同时又更加通用化、可伸缩、可扩展。

描述你想要什么，而不是怎么做

用户故事比需求文档或用例拥有诸多优势，其中一个就是把关注放在了构建什么而不是如何构建上。开发者通常把需求文档或用例逐字逐句地翻译成代码，使其以后难以变为通用的解决方案。产品负责人必须注意不要告诉开发者如何做某一件事，而是告诉他们要做什么。这可以给开发者更多的自由，想出更容易维护的解决方案。

及时回答问题

一句话，用户故事不能取代需求文档。用户故事用来引导产品负责人和开发者之间的对话。产品负责人必须时刻准备回答开发过程中产生的问题。通常回答开发者的提问会成为开发阶段的瓶颈，当产品负责人无法响应的时候，开发过程就会变得缓慢，开发者必须对他们所做的事情做出结果未知的揣测。

消除依赖

产品负责人通常不编写代码，但他和开发者依赖的其他团队协作，保证他的团队不被依赖所累。产品负责人排定待处理任务的优先级，保证团队里所有的依赖都有足够的预留时间。

支持重构

产品负责人的职责是提出需求，但他同样也必须对生产出来的代码的质量保持关注，以保证代码的可维护性和可扩展性。这通常意味着在团队认为他们需要重构的时候给予支持。

在Scrum中，产品负责人是推动产品开发至关重要的角色。他并不是个技术角色，但需要有很好的天赋和沟通技巧。当产品负责人一直能够迅速回答问题并指导方向的时候，软件开发才能迅速地向前推进。

5.7.2　编写出更好用户故事的 7 个策略

用户故事帮助人们关注构建的是什么以及给谁构建。下面是编写出更好用户故事的7个策略。

把它当作一个占位符

用户故事无法取代需求文档。它帮助产品负责人和开发者展开对话。这些对话取代了需求文档，用户故事仅仅是占位符。使用用户故事来把控你想要放进冲刺计划里的想法，便于以后进行讨论。

关注“什么”

用户故事关注一个功能做什么，而不是如何做。如何构建一个功能应该在开发者编码时决定，但首先需要确定功能是做什么以及它会被如何使用。这帮助开发者隐藏实现，让软件耦合更少、更容易扩展。

把“谁”人格化

知道一个功能为谁而做能帮助开发者更好地理解这个功能如何被使用，给开发者优化设计提供灵感。它同时也帮助开发者围绕用户需求组合功能，对特定类型的用户构建更完善的功能集合。给你想象中的用户一个背景故事——他或她的名字是什么，有什么愿望，有什么兴趣，等等。这会帮助你更好地想象和理解将来会使用这些功能的用户。

知道为什么会有一个功能需求

理解为什么会有一个功能需求和它想要达成什么，这通常会让我们想出更好的方案。用户故事中“所以从句”通过陈述一个功能带来的收益定义了为什么会有一个功能需求。只要遵循这个功能的核心需求，就能给我们的开发过程带来更多选择。

开始时简单，日后再加强

增量式设计和开发不仅是最有效的构建软件方式，也会带来最好的结果。演进式设计通常更加准确，更容易维护，更容易扩展。重构和演进式设计能帮助我们更快速构建更高质量的软件，使我们能够以最少量的重复工作来修改设计。

5

心系边界情况

用户故事陈述了**快乐路径**，但我们也需要考虑许多其他路径，包括次要路径、异常路径、错误路径。我通常把边界情况记录在用户故事上以便跟踪，之后我会根据这些编写测试，然后用测试驱动实现。

使用验收标准

在开始实现用户故事之前，很重要的一件事是确认验收标准。最好是通过一组验收测试来描述，要么使用验收测试工具（例如SpecFlow、FIT或Cucumber），要么写在用户故事卡片上。验收测试告诉开发者什么时候实现了一个用户故事——当所有的验收测试都通过的时候。这帮助我们远离过度开发。

用户故事从本质上和其他的需求文档不同，它是关于一个功能是什么、为什么和给谁做的最小描述。这足够引发产品负责人和开发者之间的对话，这样开发过程变成了一个探索式的过程，而不是开发者盲目跟随需求。

5.8 总结

在问如何做之前先问做什么、为什么做、给谁做是为了描述目标和限制，而不是实现细节。讨论构建什么而不是如何构建，让开发者发现如何构建并且把构建方式和构建意图抽象化。这帮助我们隐藏实现细节，让代码更容易维护而且扩展成本更小。当用户故事取代了需求文档并且开发过程变为探索式的时候，我们构建出的产品会比试图提前预估一切的时候更好。

本章中心思想如下。

- 通过关注软件应该**做什么**而不是**如何做**，开发者可以自由地探索最好的实现方式。
- 为了构建更优质的软件，你需要知道如何和你周围的人沟通。
- 把定义功能的关键性对话从描述实现细节转变为描述**做什么**、**为什么做**、**给谁做**，这有助于建立探索式开发流程。
- 优秀的产品负责人编写出高质量的用户故事和清晰定义的验收标准。
- 消除需求文档的编写，在产品负责人和开发团队之间建立创造性的合作关系，构建功能将更高效，可以节省三分之一的开发时间。

把定义功能的方式从描述实现细节转变为描述做什么、为什么做、给谁做，消除需求文档的编写，在产品负责人和开发团队之间建立创造性的合作关系，构建功能将更高效，可以节省三分之一的开发时间。

第 6 章

实践2：小批次构建

Scrum中的时间盒子概念促使开发者把大的功能拆分成小的任务，虽然听上去简单，但实现起来却挑战十足。开发者习惯了一次构建若干个功能，关注的是最终发布而不是单一的功能或者任务，但这样的话他们可能会应接不暇、进度落后并且急于求成。

电影制作者并非用一两个小时拍完整部电影，通常是一个场景一个场景、一个镜头一个镜头拍出来的。拍完一个镜头之后开始拍另外一个。虽然导演时刻着眼于整部电影，但是那一天或者那一天的一段时间内的全部精力都放在**那一个镜头**上。

有些时候，电影制作者会运用多个镜头和其他技术试图一次拍摄整个场景。同样，软件项目中的某些任务会比其他的任务更加复杂，所以每个软件开发相关的人都需要更好地理解，哪些任务是“镜头”而哪些任务是“场景”。

在软件开发中，有许多因素决定某一工作单元的大小，“小”并不是唯一标准。这一工作单元应该可以展现出**可度量的结果**，应该具有一些可以被人观察到的行为。

有时候需要不少的工作才能展示一些可观察的行为。有一些因素促使我们把任务变得更大，有些因素促使我们把任务变得更小，我们需要在两者之间找到适当的平衡，才能让任务**大小适中**，而不仅仅是“小”而已。

当团队需要在一个较短的期限（比如几周）内构建有价值的软件的时候，会促使他们把大的问题拆解成更小、更容易管理的问题。这样做需要技巧，当一个团队熟练掌握这些技巧后，会发现他们不再需要严格地依附于迭代周期了，开发过程变得更流畅。

时间盒子是一个用固定的时间周期执行任务的实践方法。范围盒子则不管时间而倾向于一个可以被简单描述的完整的工作：一个用户故事或功能。在时间盒子和范围盒子之间进行选择，取决于工作的类型。如果任务大小都是统一的而且比较小，范围盒子比较适用。管理范围比管理时间更需要技术，所以通常先选择时间盒子，直到你已经可以熟练地把功能分解为较小的任务。

我们用不同的方式寻找一组工作的适当大小，但首先应该停止自欺欺人。

6.1　更小的谎言

事实上，我们习惯性地自欺欺人，这其实是件好事。它给生活以确定性，而事实上生活中没有什么事情是百分之百确定的，如果我们完全接受这个现实，就没人愿意早上起床面对人生了。生存本能要求我们带着确定性生活……但并非**总是**这样。

如果我们必须用谎言自我欺骗才能工作——所谓“谎言”绝非贬义——那么让那些谎言变得小些，好让我们不至于在真相面前太过痛苦。这其实是敏捷的本质。我们设立更小的目标，构建更小的系统，好让我们在觉得偏离方向的时候可以尽早做些调整。这是至关重要的一部分：**做出调整**。

仅仅知道要做什么——做正确的事情——并不够。我们中的相当一部分人知道要做什么，却因为种种原因没有真正放手去做。

相对于用独立的分析、设计、编码、测试、部署阶段来构建软件，逐个功能地构建系统，即每几周往一个正常工作的系统中添加功能要简单得多，而且风险小得多。这样做更为简单是因为小的任务比大的任务更容易执行，而且风险更小，因为功能构建完成之后就会被集成到一个正在工作的系统中，所以没有任何意外。

开发者不再是在漫长的开发周期中用大的谎言欺骗自己，而是在“两周迭代”中用小的谎言自我欺骗。

6.2　学会变通

项目管理并非因为其灵活性而为人所熟知，但现实往往迫使我们不得不变通。我们要不准备好去改变，要不被它迎头一击。

也许你会对工作范围做出变通，也许你会对发布时间做出调整。但事实上，我们有时候必须要对**两者**都做出妥协。注意，我们说的是“范围”和“时间”，因为“资源”二字并不适用于人类。

人类无法度量。

Frederick Brooks在《人月神话》[Bro95]中详述了关键路径的重要性：“怀孕生子需要九个月，无论有多少妇女参与其中。”一个妇女九个月可以生产一个婴儿，九个妇女一个月不可能生产一个婴儿。

如果你生产的是烤面包机，拿到了一个大订单，想要加倍生产，你可以雇用双倍的员工，加入一条新的生产线，然后双倍地产出……这样是行得通的！

但是如果你是在构建软件，有大量的需求，想要产量加倍，如果你投入双倍的人力……会发生什么？

事情会变得更缓慢，甚至完全停滞。

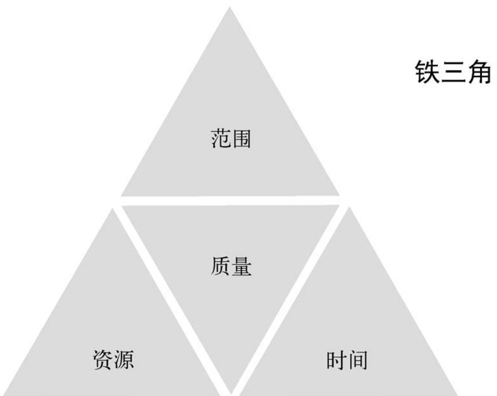
这是软件开发和制造业完全不同的又一佐证。

制造业有并行的规则——两组完全独立的生产线可以让产量加倍，但是在软件中则有许多人与人之间的交互，在增加更多人手的时候，交互变得更复杂而速度则变得缓慢。生产线的机械性工作可以完全独立，但是软件开发却不是机械性的工作。通常，当我们在项目中加入更多人手的时候，需要更多的沟通和协作，这会让项目开发速度变得更慢而不是更快。

事实上，软件行业中的顶尖团队的秘密之一是他们都非常小。之所以这样，是因为人类之间的交流非常耗时间。

所以如果你不能增加更多的人手，而且因为交付日期的限制而没有更多时间，但市场人员要求功能必须交付，那么什么会被舍弃掉呢？

开发者知道是什么。是唯一一样还在我们控制当中的事情：**我们工作的质量**。但是质量是我们从来都不想牺牲的。



这就是所谓的铁三角[1]：范围、时间、资源（scope,time,resource，STR）。在制造业中，人们相信确定了其中两个就能够得出第三个。STR 遵从如下关系：

$$范围 = 时间 \times 资源$$

但是对于软件工程来说，这是个错误的公式。在建筑构造中，范围必须是固定的。你不能“发布”一个没有封顶的建筑，甚至那都不应该是最后一步。

在软件中，范围是**最容易**变通的。开发者经常会构建错误的内容或者对正确的事情过分开发，所以可变通的范围应该是我们首先要理解的。理解如何使用诸如“最小可市场化功能集”（minimal marketable feature set，MMF）、“故事分割”等技术来控制范围，可以帮助我们调整出大小得当的

① 亦请参考 Project Management Triangle，http://en.wikipedia.org/wiki/Project_management_triangle

功能集。

最有价值的功能应该首先构建，如果用户可以从中得到价值则尽早发布给他们。这样，如果在最终交付日期到来之时开发进度并没有预期的快，至少软件可以产生一部分价值。鉴于将近半数的交付功能从未被使用，那么交付给用户部分功能和完全没有交付之间的差别就是成功和失败的差别。

所有的这一切都指向更小的反馈周期。得到的反馈越多，就越容易发现问题，而越早发现的问题就越容易处理。

通过小批次构建，我们寻求的是验证而非假设。

这同样也是思维上的转变，现在你开始思考在开发时有哪些依然是**未知数**。

6.3　控制发布节奏

在“发布节奏是如何决定进度的”[1]一文中，Mary Poppendieck探讨了发布周期如何影响系统的开发效率。她以6个月的发布周期为例。如果你的发布周期是6个月，你自然会花费至少1个月在需求上，判断出需要构建哪些功能。这些时间没有花在编码上。同样，我们差不多需要花最后的2个月甚至3个月在测试和集成上，这几乎放之四海而皆准。

这意味着开发者有超过一半的时间**没有花在编写软件上**。

假设你可以在6个月的时间内构建25个功能，基本上一个功能1周。假设这些功能的需求随机出现，在下个发布周期之前平均等待时间是3个月（发布周期的一半），加上额外的6个月完成发布。这意味着一个功能的平均等待时间是**9个月**，而你的效率只有2.6%。这样的发布效率低得惊人。

假设所有的任务都需要1周完成，则发布周期越短效率越高。

发布周期	平均等待	效　率
1 周	1.5 周	100%
2 周	3 周	50%
1 个月	1.5 个月	23%
2 个月	3 个月	12%
3 个月	4.5 个月	7.7%
4 个月	6 个月	5.8%
6 个月	9 个月	2.6%

① Poppendieck, Mary. How Cadence Predicts Process (blog). http://www.leanessays.com/2011/07/how-cadence-determines-process.html

当然，不是所有的任务都能1周完成，而且任务大小的差异越大，我们的发布周期就越长。“效率”一词在此处可能会有歧义。它表示软件开发过程中在单一任务上耗费了多少时间。半成品越多，总体上系统就越低效，同时也需要更多的任务切换。

发布周期越长，这样的低效被放大得越厉害。长的发布周期必然需要编写需求文档，效率就更低。它也会影响正式的测试和集成阶段。

长的发布周期迫使我们回到瀑布模型。

发布周期越长，就会有越多的任务会被按照类型分组（设计、编码、测试、集成），而不是每个任务尽可能快速、高效地从头到尾完成——从用户故事直至集成。

成组的功能和长的发布周期需要编写需求文档，这会造成大量错误加入并导致低效。根据IAG 咨询公司的“2009 年商业分析报告”①，新项目中超过41%的开发资源被不必要的或低质量的需求文档所消耗。

敏捷以用户故事取代了需求文档，我们建立用户故事是为了**引发对话**，所以敏捷的真正目的是为了用对话取代需求文档。你依然知道应该构建的内容，而不是客户想要的特定功能，那是需求文档真正应该做的。对话需要以一种有**准备**的方式持续进行——经过事先准备的对话能够保证你对所构建事物的真正关注。

和其他人交谈，和对方面对面提问，“这种情况下我应该返回正确还是错误”。然后得到回答，“嗯，咱们讨论一下”。这样的效果比根据需求文档编程好得多，因为需求文档根本不会引发那样的提问。于是开发者就自己猜测，对一个开发者来说，能有**一半**的时候猜对就算很幸运了。

你愿意在掷硬币上下多大赌注？

6.4 越小越好

将大任务拆分为小任务需要一定的技巧，即在分割的同时保持代码的模块化。

但我必须说明：当我说拆分为“小任务”时，并不是指盲目地把任务拆开然后再合并成一个大的功能，这样并不管用。如何拆分大任务是所有开发者和产品负责人都需要掌握的关键技巧。

当团队关注于功能的时候，他们会说：“我们想看到系统的某一行为，而且希望该行为大到可以被观察到，但又不至于大到难以测试。我们在接下来的**2周**内可以做什么？”

但是在瀑布模型中，团队成员不清楚如何对工作进行考量。我这周的工作量是多少？如何根据结果评估我的工作？在瀑布模型环境下的工作量通常是“我写了一堆代码”。尽管存在任务分解，但是仅仅是把功能分解为函数而已。

① IAG Consulting. “Business Analysis Benchmark.” http://www.iag.biz/resources/library/business-analysis-benchmark.html

通常情况下任务分解的工作不是由开发者执行的，一般是由架构师或者主管人员将分解好的任务安排到开发者手上："编写这些函数。"然后开发者编写这些函数，接着编译它们，也许一步步调试，然后这些函数被放到队列中等待最后的集成。这个开发者的工作是根据代码行数进行评估的——但客户丝毫不关心这些。客户并非根据代码行数评估软件：代码行数对客户一点用处都没有，向客户交付更多行代码毫无意义。

那么我们的目标是什么？我们又应该用什么来衡量自己呢？

回想一下W. Edwards Deming的教导和精益思想。我们应该用**对客户的价值**来衡量自己。这是我赞同的为数不多的几个度量方法之一，因为它不鼓励局部优化。几乎所有其他的度量方法（代码行数、速度等）都是局部优化。如果客户价值以外的其他东西拖了你的后腿，那就是没有价值的，没有益处的。

越小越好有四个基本原因：

- 更容易理解；
- 更容易预估；
- 更容易实现；
- 更容易测试。

更小的任务有更小的风险，因为获得反馈的机会更多。

6.5　分而治之

分而治之策略对马其顿国王菲利普二世很有成效。让敌对的城邦互相争斗，所以永远不会同时与几个（或一个）对手正面对决，不用和联合势力抗衡。这个例子告诉我们，大的问题仅仅是许多小问题的集合，而小的问题比大的问题解决起来容易得多。所以本质上就是**问题分解**，这是一种技能（并不是唯一的技能），而且是软件开发从职业到专业需要发展的关键技能之一。

海地人有句谚语：**小蛇要在躲藏中成长**。

如果你希望除掉家中的蛇，要趁它小的时候下手。不要等到它长大了，需举全村之力才能除掉它。要把毒蛇扼杀在摇篮之中。

如果你在初期就进行处理，许多问题都是很容易解决的。这并非火箭科学那般高深。我们在这里讨论的每件事情都是基本常识。但常识和传统观念常常相互矛盾。

典型的用户故事、功能、任务，这些我们必须处理的事情，之所以会很繁重，是因为两个原因：要么是**复杂型**的，要么是**复合型**的。

把复合型用户故事拆解为不同的组成部分。如果一个用户故事是复合型的，那么你很容易就能识别出有哪些组成部分。针对每一个部分分别建立独立的用户故事即可。

如果一个用户故事是复杂型的，那么通常只有一个原因：其中有许多未知因素。我们处理复杂型用户故事的方式是“分离已知和未知”。我们不断地在未知的范围内进行迭代，未知的范围越来越小，直至消失。

再强调一次，敏捷方法里面的“时间盒子”非常有价值。比如说：我会在下次迭代中研究这个问题，然后找出有什么方法可以解决它。有没有一些库函数可以帮我解决问题？我能否把这个问题简化？我需要知道哪些关键点？我还有哪些不清楚的地方？等等。

探索未知事物的时候需要做两件事。

- **把未知变为已知**，所以我们知道如何处理大的未知问题。
- **把未知进行封装**。如果能把大的未知问题**藏起来**，那就藏起来，以后再处理。

如果一次处理太多的问题，我们就会浪费大量时间，因为有许多半成品。这是队列理论背后的概念，瑞士敏捷顾问Hakan Forss用队列理论来管理公路交通模式。正如在某些交通瓶颈处降低速度限制会让更多的车通过一样，“降低并行任务数量可以使系统更稳定”。[①]

6

Forss使用了下面的利特尔法则：

周期时间 = 任务数量 / 吞吐率

任务数量（待办任务列表中项目的数量）除以吞吐率（单位时间内所处理的任务数量）就得出周期时间。

减少待办任务列表中项目的数量，周期时间也会相应变短，进而得到更快的反馈，可以尽早发现还处在萌芽阶段的问题，修复起来也更容易。

在传统的需求文档驱动的开发流程中，**每件事情**都是提前加入待办任务列表，待办任务列表就是需求的代表。这产生了巨大的任务数量，然后让开发周期变成数月甚至数年，小的问题随着时间的延长而更加恶化，演变成系统崩溃级别的bug。

不仅仅是在瀑布模型软件开发中才这样，在很多声称实行敏捷的组织里也如此。只要你把集成和测试阶段放到后期，半成品的数量就会很多。一个完成度99%的任务没有多大价值，因为其风险未知。消除某项功能风险的**唯一**方式就是当它**开发完成**时就把它完整地集成进系统。

正如我们所见，解决方案就是持续集成。持续集成对是否引入bug提供即时反馈。此类即时反馈帮助开发者在缺陷还比较小的时候就及时处理，所以不需要太多功夫便可以继续其他的工作。

① YouTube. “Hakan Forss ‘Queuing theory in software development.’” Uploaded July 1,2011. https://www.youtube.com/watch?v=tt4vnCzHAZk

6.6 更短的反馈回路

当我关注反馈回路的时候，我更倾向于小的批次。我指的是软件开发的**所有**反馈回路，反馈回路可能会有**多个**。

和客户交流是一种反馈回路，在迭代或者功能开发的后期进行演示也是一种。软件开发者也可以从编译器得到反馈回路。我们编写一段代码然后编译，编译器告诉我们有哪些语法错误或者其他问题——这是软件开发者得到的最基本的反馈。下一步是从自动化测试套件中得到反馈，测试套件可以即刻发现代码错误或者其他问题并且通知我们。

但是仅仅拆分任务和增加反馈还不够。软件开发者需要得到**建设性**的反馈，好让他们**做出响应**。

反馈可以是好事也可以是坏事，取决于你是否可以针对反馈做出调整。我们需要找到收集和处理反馈的方法。敏捷理论使用“回顾会议”“代码审查”“问题登记”等方法。有许多得到反馈的方式。但是也许对于开发者来说最重要的是，要有一个快速自动化构建过程，让我们可以依靠它随时来捕获错误。

我认识的一些顶尖的软件开发者在编码的时候**每二十秒**（一分钟三次）执行一遍测试。这是一种优秀开发者都会学着接受的辅助工具。

软件开发是一种智力密集型专业，一旦步入四十岁阶段，我们就知道自己反应不如从前了，所以必须依赖其他的技巧。我坦然地教授软件开发者如何变懒，因为我们越懒越**深思熟虑**。而且这样可以得到更好的结果，承受更小的压力。

在讲课的时候我尽量不使用“错误”这个词语，因为对一个专业的软件开发者来说没有绝对的对错——只有权衡。但是，有位学生在一堂课上说了一些我认为是错误的事情，当我指出来的时候，他说：“告诉我怎样可以一次做对。”

然后我说：“恕我直言，这种问题本身就是错误的，因为这是**自寻烦恼**。”

强迫自己一次做对实际上是非常困难的。如果你允许自己犯错，但了解其代价并控制错误的影响，才是最快捷的路径。两点之间的最快路径不总是笔直的。

我希望客户的问题尽快得到答复。我们都反复地听说过，许多组织说他们无法承受客户长时间驻场的成本。而我认为，你无法承受**不**这样做的成本。如果没有客户在场，开发者开始自己猜测，无论开发者多聪明也会有半数情况猜错——又回到了“你愿意在掷硬币上下多大赌注”的问题。

软件开发者经常忽略另外一种绝对重要的反馈回路，即真正提高构建速度，它成本低廉、回报丰厚。

6.7 提高构建速度

当我说“提高构建速度”的时候，我指的是，**建立起一套构建系统，在几秒内而不是几个小时内给出结果**。

为了得出这个结果，你必须知道你有哪些依赖。很多时候这个流程可以让开发者知道他们的架构是否优秀。

找我咨询的客户希望我快速给出答案，所以我学习了很多快速产生价值的技巧。我经常给他们关于如何安排工作重心的建议，所以我花了大量的时间在代码上。而构建过程则可以让我知道他们架构的优劣。

我有个客户，他们每个开发人员的机器都需要24GB的内存，因为即使是修改了消息窗口中一个单词的拼写，也需要重新构建整个企业级系统，系统内部毫无封装。他们的架构很优秀，唯一的问题是架构没有封装。开发者认为：“我不需要用这个API。那不过是改变数据库中的一些数据而已。我可以自己做。”

因为这些开发者不守规则而且没有强制要求，所有事情都乱作一团。结果就是，需要每台机器都有24GB的内存去构建。

但是没事，金钱成本很低。多花钱在硬件上就解决问题了！

这也许是最后的手段，但他们现在需要三周的发布周期——花费三周的时间去验证一个待发布版本是否可以发布，即使是一个很小的改变，因为他们需要手动重复测试所有功能。他们希望改成双周迭代。算一下就知道根本行不通，所以他们必须重写一切。

让我再说一遍：**他们必须重写一切**。

必须重写整个系统的情况很少见，但是也会有，上面的例子是因为系统的行为和数据都一团糟。而且，他们的生意每年以十亿美元计，所以不是一般小公司。他们艰难地让所有客户都转向他们的新系统。他们处在难以想象的困难境地。

软件开发者没有“买来不改就能用”的现成组件。我们不是购买各个组成部分然后组装在一起构成软件，但我们可以分组进行构建。如果你的车因为火花塞不打火而无法启动，但火花塞又是焊上去的，连接着十几个其他组件，而且同时用来支持引擎组件，那么更换火花塞或许还不如换辆新车来得划算。这就是我们所谓的“依赖”。

优秀的软件开发实践需要在构建不同组成部分的同时保证软件的完整性，而且需要让各个部分尽可能地独立。

我们会发现有许多错综的枝桠，那些不同事物间的联系，而那些联系反映了真正问题。举例来说，客户和他的地址存在着联系，这是正常的联系。客户的邮政编码和配送者的邮政编码也存在联系，但那不是正常的联系，因为如果只有一个配送者，他需要给包含各种邮政编码的所有地

址配送。所以必须弄清楚我们可以将哪些东西解耦（那些可以从其他元素中剥离的元素），哪些是不能解耦的至关重要的联系。

这在真实世界中是常识。我们想都不用想就知道火花塞不应该永久性地连接在引擎上。在虚拟世界中则不是那么明显。在虚拟世界中我们不习惯用组件的方式思考。

我并非鼓吹完全没有耦合，我推崇的是适度耦合。何为适度又是因人而异的。通常当你用不同方式思考问题的时候，你得出的解决方案也是不同的。软件也是如此。如果你有很多解决问题的技巧，那么就在你的工具箱里面一直翻找直到找出最合适的方案。

木匠也是这么做事的。木匠的工具箱里有许多工具，优秀的木匠知道何时、如何、为什么使用每一件工具。软件开发者和手艺人、学者、艺术家一样，需要知道自己有哪些工具以及它们各自的适用范围。

6.8　对反馈做出响应

我记得有一个身负重任的团队负责人，在一次例行的关于客户和团队的审查会议中，他被问到："你们最重要的提议是什么？"他回答："我们要买一个咖啡壶。"

咖啡壶在四楼，他的团队在三楼。团队中每个人都喝咖啡。他做过计算，咖啡壶在四楼，每年要花费四万美元的成本。于是他们给了他二十美元买了个咖啡壶。

作为一个专业开发者，工作的时间越长，我就越会从商业角度来看待问题。在商业中，我们需要关注两件基础性的事物：价值和风险。这其实是商业的核心理论，而且是一个商人的核心词汇。

软件开发者的词库要大得多。软件开发者使用的词汇和商人不一样，所以商人通常不理解软件开发者对同样事物的描述。

提供持续反馈的系统只在对反馈积极响应的文化氛围当中才有效。在有些组织中，预算、交付日期和功能范围在项目一开始就已经锁定，就好像在说："如果这辆列车正在驶向悬崖而我们又无能为力，不要告诉我。我宁可在我生命的最后时刻因无知而快乐。"

但如果有时间和意愿做出反应，那么我就**想要**知道面临着什么，好让我做出应对！这就是寻求反馈的最初目的——做出应对。

假如每两周的迭代都有2%的进步，那么一年下来你就进步了50%。所以无论多小的改进都值得寻求。

精益创业运动为了探索市场真正需求而生。他们认为利用精益创业的公司可以通过"利用工具持续对愿景进行验证"建立秩序而非混乱[①]。但是并不仅仅是为了创造更好的捕鼠夹子，我有

① The Lean Startup. "Methodology." Accessed November 12, 2014. http://theleanstartup.com/principles

更好的捕鼠夹子又如何呢？[①]

没人在意这些，因为捕鼠夹子并非人们日常所需。

在软件中，有许多方法进行验证和测试，正如在精益创业中一样，我们推崇持续性测试。在网络世界里工作，让我们了解到两件重要的事情是违反直觉的。一是你可以从很小一组人（一千人或一万人，甚至可能是一百人或更少）的样本当中看到趋势。二是不仅可以看到趋势，还可以找到更好的方案。你能知道用户是否购买你的产品。我们称之为A/B测试，在A/B测试中你将只有一处不同的两个版本发布给不同用户进行测试。它的效果令人难以置信。

一位来自谷歌公司的开发者告诉我，他们A/B测试了一个组件中的单像素分割符和双像素分割符，从而发现一个像素的差别在响应上有着17%的巨大差别。

谷歌公司A/B测试所有的事情，他们有专门的算法整合数据、做出判断、给出最佳解决方案。这并非难事，但他们如此坚持得到了非凡的成果。

6.9 建立待办列表

待办列表基本上就是我们需要构建的用户故事的列表。用不同的形式整理功能（用户故事），可以通过主题、用户类型、目的或对你来说有意义的任何其他方式进行整理。本质上，你是在汇集功能以便发布产品。这称为“最小可市场化功能集”（minimal marketable feature set，MMF），它代表了一个产品若想可用并且有价值所**必需**的东西。

如果知道 MMF 中有哪些功能，可以加入一些非核心功能到发布版本中，以便在开发提前完成的时候有事可做。如果时间紧迫，也能知道哪些功能关乎产品生死，哪些功能是为了给用户提供价值而必不可少的。

讨论待办列表的**顺序**而非**优先级**。产品负责人是唯一决定下一个构建什么的人，而那就是下一个最重要的功能。有时候，“什么是最重要的”并不像每个人认为的那么显而易见。有时候需要先构建第二重要的功能，因为其中有最重要功能所需的一些特性，这样做起来容易得多。

让流程保持弹性。这样，团队遵循你的MMF按待办列表的顺序进行迭代，但当有人有更好的想法时，那些想法可以得到利用。

有一条我一直遵守的基本规则：如果一个功能已经在迭代中，那么我将一直坚持按计划执行；如果某个新想法可以在下次迭代或后面的迭代中完成，那么就合并进来——没理由不那么做。只要小心不在迭代中间打断开发者，别让他们正在做的事情半途而废。团队需要待办列表就是为了可以在后面加入新任务。我们用迭代的方式构建软件，有新想法的时候**可以**将其引入到产品之中。

① 西方谚语“造更好的捕鼠夹子，全世界的人就都会想来找你”，类似“一招鲜，吃遍天”。——译者注

6.10　把用户故事拆分为任务

用户故事描述了系统的一种可观察的行为，但可能过于复杂或庞大而无法在双周迭代中完成。这时需要把用户故事进一步分解成一般性工作项目，我们称之为**任务**。

理想的任务是需要大约四小时完成的工作，但是通常由于种种原因，可能需要几天甚至数周来完成任务。

城市居民知道，多数的任务（即便是小的任务）也需要至少四小时完成。如果你想买双鞋，或者出门做其他事，通常至少需要四小时。我在考虑任务的时候总会想起这些。

软件开发中几乎所有的任务都需要至少四小时完成，即使是最小的那些，因为你必须把代码签出，必须测试，必须把代码签入：你需要做一系列这样的事情。

我们不使用小时进行估算，我们使用故事点，有些人称之为**理想工作小时**。在一个八小时的工作日里，其实只有大约四小时的**理想**工作小时。所以一个四小时的任务需要一天的工作时间。这就是能划分任务的最小单位。

归根到底，百分之百利用率的高速公路会是什么样子的？百分之百利用率的高速公路会跟停车场一样。员工也是一样。当你让一个员工以百分之百的负荷工作的时候，他通常会停摆。百分之五十左右是理想的负荷。如果你的开发者能够在百分之五十的时间里都产生实际价值，那将是非常惊人的。人毕竟不是机器。我们需要时间持续学习，我们需要伸展身体，我们需要吃饭，我们需要上洗手间。

我们都是凡人。

对于任何管理者来说，最重要的问题是：你的开发者有多少时间花在**开发**上面？

在多数环境下，开发者通常花费三分之一到四分之一（有时候是五分之一）的时间在实际的编码上。但是，开发者是被雇来编写代码的，不是吗？那不是开发者**最喜欢做的**事情吗？

每个任务都让我们距离最终功能更进一步。把任务通过验收标准来分解，让我们可以评估或观察到任务的完成情况，如果我们距离完成整个用户故事（功能）更进一步，我们就知道方向是正确的。有许多方法帮助我们这么做，正如用户故事一样，任务需要结果导向。

6.11　跳出时间盒子思考

Scrum以及其他敏捷方法传播的速度很快，这是好事，但是它们的本质却没有被广泛理解。面向对象编程（Object-Oriented Programming，OOP）从1990年就开始流行，几乎所有人都在用诸如Java、C++或C#等面向对象的语言编程。但是，如果你深入他们的代码就会发现，他们并没有真正使用面向对象编程：他们没有真正理解面向对象的语言相对其他语言的核心优势。他们并没有用对象封装行为、增加可测试性以及对设计进行解耦……只不过是用 `class` 语句包裹过程

化代码。

在敏捷中也是如此。2013年6月发布的“Scrum状况”显示，40%接受调查的公司——我认为这覆盖了整个产业中很大的范围——部分采用了Scrum。但是当你问他们使用了哪些敏捷方法的时候，却发现只用了“站立式会议”和“迭代”，而且并没有把迭代开发进行到底。这些组织中只有13%用到了持续集成，而其中只有37%每天或者更频繁地使用持续集成。换句话说，实践Scrum的公司中只有不到5%真正每天都集成他们的代码。他们把尚未完全完成的任务推进到时间周期的末期，然后进行测试。

这依然是瀑布模型。

你会看到一些收益，但没有真正的改变。Scrum并不是一个“全有或全无”的提议。有些Scrum方法论中的元素可以在特定范围有所帮助。但对于风险来说，则**是**“全有或全无”。两者必居其一。要么有风险，要么没有风险。当你有风险的时候，则充满未知。未知即是风险。

6.12 范围控制

如果你有1%的潜在风险，那么你就有很大的风险。降低风险的唯一方式是把用户故事进行到底，这意味着我们需要对什么是“完成”有明确的定义。当你把迭代完成的时候，当系统完整地集成的时候，你就知道没有风险了。在那之前，你心中都会有个问号。

这个概念让人想起“薛定谔的猫”[1]。

奥地利物理学家埃尔温·薛定谔提出了一系列的思想实验，用来解释量子力学。他假想（并没有真正建造）一个盒子里面装着一只猫，里面有一个装置，当猫有意或无意触发这个装置的时候，就会释放某种毒素。他指出，在不确定的状态下，无法知道毒素是否被释放，而且无法观察到猫的状态。我们必须接受，在未知的状态下，猫既是死的也是活的。唯一确定的方式是打开盒子。

在多数的软件开发项目中，我们的“猫”（系统）一直和未知数量的bug一起处在盒子中。在我们把程序从盒子中取出看到其执行之前，它都和薛定谔猫一样既是活的又是死的：系统既能正常工作又不能正常工作。

风险倾向于几何式增长，直至一个bug让整个系统崩溃，所以你应该保证系统的每个部分、每个迭代或每个功能都和系统完整集成以消除风险。

但事实是，了解**如何**分解任务并不那么容易。Scrum要求双周迭代，为了这样做，任务必须被增量分解到双周迭代中去。把用户故事分解成任务是一项技术，和其他技术一样，软件开发者必须熟练掌握这项技术。

① http://en.wikipedia.org/wiki/Schr%C3%B6dinger%27s_cat

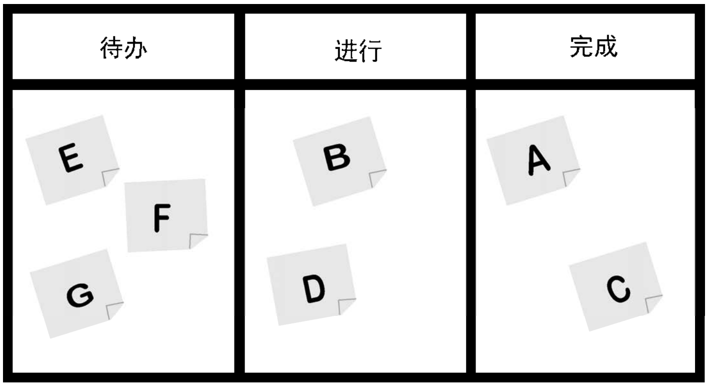
如果你习惯更严格的流程（诸如“时间盒子”或“范围盒子”），学习它们，成为习惯，理解其中的技艺，建立一组守则和流程，然后进行增加和调整，最后你可以移除“时间盒子”进而构建更小的任务。

这全都关乎习惯。如果你想戒烟，你可以用尼古丁贴片来戒除尼古丁。但是如果下半辈子一直使用尼古丁贴片，也许口气会比以前清新，肺功能开始恢复，但你并未戒除尼古丁：你依然对这种药物上瘾。

极限编程和Scrum都是和尼古丁贴片类似的东西。“迭代”真正的目的是让团队消除以发布为单位的构建习惯。一旦消除这种习惯，你就不再需要“时间盒子”了——正如吸烟者最终可以摆脱尼古丁贴片过上健康无尼古丁的生活。也有一个敏捷方法用来做到这点，称为**看板**。

看板要求我们限制正在工作的项目数量，限制每个队列（待办、进行、完成）的大小，但没有冲刺概念。项目可以从“待办”移到“进行”，完成之后移到“完成”，或者由于新的优先级而移回“待办”。

但某些项目必须移回“待办”以空出“进行”队列的空间。否则，最后会变成同时处理所有的事情。这一切的目的是为了让你工作起来更方便，而不是更困难，试图同时处理所有的事情要**困难得多**。半成品限制了一个团队在规定时间内完成任务的数量。



看板

归根到底，我们关注的是对范围的管理，我们用时间来管理范围。双周迭代是人为规定的，多少有些低效。但这是个会让你变得更高效的途径，最终你可以做到“我有个任务，我觉得需要四小时完成”，然后你开工，完成，接下来问“下个任务是什么”。这时候，你的效率就达到了全新境界。

双周迭代来自于大部分人觉得舒适的时间——转变起来最容易的时间。记住，让你构建更大的东西的时候，压力也会更大。你想要这个功能有这样的特性。把事情分解很难。波音公司没法只建造一只机翼就开始测试飞机。他们必须建造两只机翼。

团队需要有固定的汇报周期，坐下来看看他们的进展如何。这和其他会议一样，不能为了开会而开会。这是一个机会，用来查看是否有阻碍、工作进展如何，并跟进项目进度。习惯成自然。

我喜欢看到团队工作流畅而且富有创造性，为了这样你必须设定节奏。双周迭代的另一个好处是可以保持节奏。需要和开发部门保持同步的其他部门，可以每两周和开发部门进行同步。即使在一个完全的流式环境中，依然需要同步点，否则其他部门没法和开发部门交互。

所以，我们的最终目标是更小的任务。把功能分解成最小的任务，可以在四小时内完成（越小越好），应该有定义良好的验收标准：对“完成”的清晰定义。

这样就行了。

6.13　让我们付诸实践

以下是把这些想法付诸实践的方式。

6.13.1　度量软件开发的 7 个策略

软件和现实中的商品不一样，所以度量的方式也不同。软件开发者每天进行的任务都不同，所以没法用常规方法直接度量生产率。我经常建议团队不要用速度来度量，因为那样会传递错误的信息，给管理者错误的目标。可以牺牲质量来提高速度，但这样做大错特错。下面给出度量软件开发价值的7个策略。

度量产生价值的时间

我们生产软件是为了满足需要，从开始构建产品到用户发现价值的时间，可以用来有效度量我们的效率。对整体进程没有影响的局部优化是没有意义的。度量“产生价值的时间”帮助我们着眼大局——值得度量的东西。

度量编码时间

软件开发者喜欢开发，不喜欢阻碍。讽刺的是，很多组织中花在保证质量上的时间会占用真正创造质量的宝贵开发时间。我知道，有些开发者仅花不到百分之十的时间在开发上，剩下的时间都花在会议、报告以及其他一些没意义的事情上。好的开发流程是，开发者可以花大部分时间在实际开发上的流程。

度量缺陷密度

多数的组织跟踪bug，但是可能会有增加bug容忍度的副作用。代码中的缺陷通常是深层次问题——开发流程缺陷——的症状。如果在产品代码中频繁出现缺陷，则意味着开发流程可能出问题了。寻找问题的根源并且修复它。缺陷密度（千行代码内的bug数量）是可以在团队间进行的少数几个度量方式之一，所以它可以用来对流程进行评估。

度量发现缺陷的时间

调查显示，修复缺陷的成本从缺陷产生开始随着时间流逝以指数方式增加。修复成本最低的缺陷是在产生之后立刻被找到并修复的缺陷。迅速找到缺陷不仅降低修复成本，而且帮助开发者从一开始就避免缺陷的产生。

度量功能的客户价值

并不是所有的功能对客户的价值都一样。事实上，软件中将近半数的功能从未被使用。待办任务列表应该被排序，用以保证最高价值的功能优先构建，不重要的功能被推后甚至取消。这让更多的时间可以花在那些高价值的项目上。如果软件开发者不能确定哪些功能的价值更高，则应该询问客户。

度量未交付功能的损失

有时未交付功能的损失是构建它最有力的一个理由。问问利益相关者，一个功能价值多少，如果没有交付的话会损失多少。答案会让你吃惊。

度量反馈回路的效率

提高效率最有力的杠杆往往是流程本身。一个优秀的开发流程中会有内建的反馈回路用来调整流程。反馈得越及时，我们的效率越高。尽快发现问题并从失败中学习，这是团队快速成长的方法。

多数团队为了度量生产率而牺牲了质量。度量生产率也许根本不可能，而且肯定会产生混乱。反之，应在交付阶段和开发阶段关注产品本身并度量它的价值。

6.13.2　分割用户故事的 7 个策略

用户故事越短越好。短故事容易预估、理解和实现。短故事有助于构建高内聚低耦合的代码。短故事更容易测试。以下7个策略帮助你把大的故事拆分为更小的故事。

把复合的故事拆分为组件

如果故事里面有子故事，则把它们分割为多个故事。这有助于解耦组件，让系统更加模块化。更小的故事也更容易开发。

把复杂的故事分割为已知的和未知的

故事之所以复杂，通常是因为其中包含未知因素。我们也许不知道客户到底想要什么，或者不知道如何实现客户所想。把已知和未知分离是分割包含未知因素的故事的第一步。

对未知持续迭代直至完全理解

一旦某些事物被标记为未知，就将其封装！通过一个定义良好的接口将其隐藏在抽象背后，你

可以自由地研究它而不会占用关键路径。通常先攻克高风险的未知，以后再处理低风险的未知。

根据验收标准分割故事

在我们把故事分解为任务之后，依然需要有一些可见的证据来判断任务是否完成。根据验收标准分割故事，有助于我们关注开发，在迭代中提供用户价值，也有助于我们明确定义什么时候一个故事算开发完成。

最小化依赖

我们希望故事互不依赖，但是依赖有时难以避免。试着通过定义良好的接口来移除组件间的依赖。如果必须有某些依赖，让后来的故事依赖于之前的故事，而不是反其道而行。

保持目的单一

一个故事应该是完成一个单一目的，或者一个目的可检验的某个方面。通常，当我们想让故事给用户提供完整功能时，会把故事弄得过大。但小的故事更容易开发。我们不需要大而全的功能，仅需要有足够让用户从中获取价值的功能。后续故事可以增强一些特性并提供更多功能。

保持故事可测试性

每个故事都应该定义一组验收测试作为验收标准。如果故事无法测试或难以测试，我们就不容易验证它们。让每个故事都有可观察的结果以便验证。尽可能地自动化验收测试。

编写故事是一个需要时间磨练的技能。当故事是高内聚低耦合的时候，系统便会更专精且容易构建。保证故事短小、专注、容易验证，有助于系统的清晰性和可维护性。

6.14 总结

小批次构建让每个任务都可以在短时间（理想情况是四小时）内完成，保证任务都满足验收标准，或者至少产生可观察的结果。这有助于简化任务，让任务更容易预估、完成和验证。

本章中心思想如下。

- 交付节奏影响着软件开发流程。
- 如何更好地控制你的时间。
- 小的任务更容易预估、测试和处理。
- 如何把功能拆分为可观察的行为。
- 通过对“时间盒子”理论的不断精进，我们得到更好的“范围盒子”理论：把任务分割为更小、更容易处理的任务。

发布周期的节奏控制着开发流程。构建可发布软件的周期越短，软件开发效率越高。通过小批次构建，我们可以保证任务更容易处理，大大减少负荷。

第 7 章

实践3：持续集成

处理痛苦的方式有两种：避免痛苦，或者学着承受。

集成代码可能会让人痛苦，暴露出之前没有发现的bug和其他问题。许多软件开发团队为了避免这种痛苦而尽可能地把集成时间拖后，最后发现在临近发布之前集成更加痛苦。磨刀不误砍柴功，这些团队给自己带来很多没必要的麻烦和风险。

如果试着适应这种集成带来的痛苦而非将其拖后会怎么样？如果每次只处理一点直到集成变成了一系列的小问题，每个处理起来都不那么痛苦，又会怎样？

“持续集成”是一种在构建时期而非发布前进行集成的实践。持续集成之所以如此重要，是因为它不仅帮助开发者尽早消除bug，也帮助开发者学习如何编写更好的代码，使其更容易集成。直到功能被集成之前，都没办法保证代码能在系统中正常工作。

持续集成作为一个反馈机制，对于开发者来说也同样非常有价值。当构建失败的时候，通常会有大量的信息，我们难以从中发现问题的根源。构建失败需要**及时**修复，所以开发者并不喜欢冗长的反馈消息，它迫使团队成员不得不在日志文件中摸索。开发者希望明确知道哪里出了问题，好立刻就能知道如何修复。

开发者应该进行持续集成，并从中了解到他们的工作对系统的影响，看看是否引入了bug，或者他们的代码是否能够和系统其他部分良好配合。

有成熟的集成工具帮助我们进行持续集成，但它们都依赖于**能正常工作的代码**。编译器会在遇到第一个错误的时候停止编译。单元测试对能正常工作的代码进行检验。调试器也需要能正常工作的代码。代码不工作，开发者只能想象如何执行程序，这往往和现实中不一样。

在软件开发的所有关键概念中，持续集成最为重要。它为所有其他技术实践提供了框架和条件。有意思的是，它也最容易实现。

持续集成仅仅是基础设施而已。所有实践持续集成所需的工具都免费，且容易实施，这些工具给开发可维护和可扩展的软件提供了前提条件。

7.1 建立项目的心跳

我把持续集成当作项目的心跳。它在后台运行，而且永不停息。它是自动的……应该自动运行。

团队大小没关系，即便只有一个人，也要买台计算机，最便宜的那种就行，把它当作**构建服务器**。

安置好构建服务器，然后等着新代码被加入到代码库中。新代码加入后，服务器会自动构建整个系统，执行自动化测试，验证各项工作，然后给出结果。那台机器默默坐在那里，一直在运行，它将一直存在……就像心跳一样。

移除在构建和发布周期中的所有人为干预，让它在后台运行，正如我们不用主动去控制心跳一样。

从大的发布周期到持续集成解决了瀑布模型项目的最大开销：验证待发布版本，而且是降低到**无成本**。

编写自动化单元测试并非没有代价，它需要时间和精力。但是一旦编写完成，就可以任意执行，一遍又一遍，没有额外开销。这意味着你可以用完全不同的方式开发：让系统本身给你反馈该如何修改系统。当开发者发现了单元测试的价值之后，单元测试就变成了一项至关重要的资源。

7.2 理解完成、完整完成和完美完成的区别

我们对流程进行评判的时候，需要做的事情之一就是定义“完成”的真正含义。通常有三种不同的定义。

完成

在传统的瀑布模型开发环境中，“完成”意味着开发者编写完了一个功能，可以让其执行，在他的机器上得到一定的结果。但是这并不够好。

完整完成

意味着代码不仅在开发机器上正常执行而且经过集成。我们能看到它符合项目的心跳节拍，能够立即发现任何潜在的异常。

而第三种则是……

完美完成

意味着代码在开发机器上正常执行，经过集成，**清晰且健壮**。这是一个在软件开发领域中经常被忽略的区域。我们绝对需要健壮性，因为这样才能花时间去清理设计和调整代码，让它们更容易阅读、更容易理解、更容易维护。这是非常重要的一步。

所以“完美完成”意味着不仅仅是正常工作、经过集成，还要容易理解、可读性强、健壮。我们都希望自己的代码具备上述所有特性，因为我们希望降低软件所有者的成本。

所以当我说代码“完成”时，我实际上的意思是“完美完成”。

7.3　实践持续部署

持续集成并不是说必须每次都部署到生产环境。你并不需要在每次有人提交代码或在每个迭代后都进行**发布**。要让市场、健壮性、可部署性、版本管理等来决定发布，而不是让开发过程决定。“持续部署”意味着你可以随时发布到生产环境。但真正发布却是一个商业决定。

所有的代码以及构建系统所需的所有其他文件都需要在版本库里维护，这样，在构建软件期间，开发者不停地把新功能签入到版本库中，所有人在同一分支上工作。这保证所有团队成员都使用同一代码库，在同样的源代码上工作。

开发者可以检出系统代码的工作副本，在开发机器上构建和运行，然后可以把他们做出的任何修改提交到版本库里。用版本库来管理所有文件，我们可以在任何时候回滚到任意时间的文件，并且知道所有修改记录。

除了源代码之外，版本库还应该管理构建所需的所有其他文件，包括配置文件、数据库模型、测试代码和测试脚本、第三方库、安装脚本、文档、设计图例、用例、UML图例等。

我曾经开设过一个课程，学员是在某互联网巨头就职的开发者团队，这个小团队在持续构建上遇到了困难。他们告诉我，他们的系统经过手工测试，但是在部署到生产环境之后得到的结果却不同，发现了他们之前测试中没有出现过的bug。我问他们是否使用了版本控制。“当然！我们把所有的源代码都放到了版本库中。”

然后我说：“那么，你们同样也把构建脚本、存储过程、数据库模型等都进行版本管理了，对不对？”

我这么说之后，有三四个人起身走出了教室。我觉得可能冒犯了他们，但下次茶歇的时候我看到其中一个人回来了。我问他有什么问题，他说他回去检查他们的团队是否把所有的构建脚本和数据库模型都进行了版本管理，事实上他们没有。

几个月后，他给我发了封邮件，告诉我这就是他们在生产环境中构建不一致的原因，现在他们在测试系统中完整地复制了生产环境。

他们之前需要用三周的手工测试周期来进行测试和发布，然后部署到两个数据中心之一。他们把大约1%的流量导入到那个部署了新代码的数据中心。如果一切正常，他们继续每次1%的导入，直到全部导入完毕，所有的用户都运行在部署了新代码的数据中心上。三周以后，他们再重复一遍上述流程，把第二个数据中心的流量导入到第一个数据中心上。

这看上去是一个挺低效的代码部署方式。

7.4 自动化构建

我希望你可以做到轻松地构建软件，轻松到几乎感知不到。你应该仅仅单击一下鼠标就可以启动构建。

漫长的测试是构建周期长的首要原因，然而有许多技术可以让测试变得非常非常迅速。构建应该在十分钟内完成。这是James Shore和Shane Warden在他们合著的《敏捷开发的艺术》[SW07]中描述的。但我认为即使是十分钟也太长了。

我希望能在一两秒内完成本地构建。在提交到构建服务器的时候，我希望在几分钟内完成构建，尽管具体时间取决于构建的目标是什么和项目的大小。有些构建确实需要几小时。但是，如果你的系统解耦合理，并不是每次做个小修改都需要构建整个系统。

如果构建时间超过十分钟，构建的频率就会降低。那样的话，找到针对特定模块的依赖，**仅仅对那些模块进行测试**。

这会让构建速度有所提高，但**真正**会加速构建的是，**编写出优秀的单元测试**，并且仅仅测试那些**需要**测试的代码，而不测试那些可能用到的代码。不用担心，我们会在第10章和第11章中讨论。

理解对于特定修改所需的依赖，不仅仅有助于找出哪些模块应该进行编译和测试，而且有助于系统的分解，让它更容易部署和扩展。应该首先在开发机器上进行构建。如果一切正常，提交到构建服务器。构建应该能够在离线状态下进行，好让开发者可以在网络出问题的情况下正常工作。

我们有针对Java、.NET和其他环境的自动化构建工具。开发机器上的本地构建成功运行并通过所有测试后，应该自动提交到版本库中，触发构建服务执行构建。一旦新代码编译完成，应该自动执行测试，验证那些变更没有影响到的系统的其他部分。执行时间非常长的测试可以放到每日构建中去。

如果代码够简单，那么测试也应该够简单。相对于进行一个复杂行为的测试，我们希望尽可能多地分解为一些简单行为进行测试，这样就能尽可能地减少那些大的集成测试。

有了如此多的加速单元测试的技术，如果使用得当，就可以在一秒内执行几千个单元测试。

构建失败耽误的不仅是造成失败的开发人员，而且是项目中的所有成员，所以修复造成构建失败的问题尤为重要。通常提交导致构建失败代码的开发者会回滚代码或者修复它。

几年前，我造访一位在开发团队中实践极限编程的朋友。他们的墙上贴满了团队协议和任务看板，每个工位上都有两台显示器、两副键盘鼠标、一台计算机，两位开发者坐在一起结对编程。看到这番景象我几乎要热泪盈眶了，尤其在我知道这个团队有多么成功之后。然后我发现在角落里有一些大功率设备闪烁着红光，上面放着一顶消防头盔。我问我的朋友那个是什么，他说是他们团队的构建服务器。然后他拿起键盘跟我说：“我来给你展示一下构建失败会怎样。”

他修改了某些代码，强制让构建失败——所有代表构建的灯都灭了，警报响起，红灯开始闪烁。

他说："在我们这里不允许有失败的构建。"

一旦构建失败，所有的开发者都会知道，导致构建失败的开发者会去构建服务器那儿，戴上消防头盔，坐在终端前修修补补。**可用的构建版本**深深植根于他们的团队文化，每个开发者都依赖于可用的构建版本，没有它开发者啥都做不了。

这个构建版本随时都可以投入运行。

7.5　尽早集成，频繁集成

集成时间越晚集成困难越大，所以我们应该随时集成。

如果集成过程没有让人痛苦的地方，则做起来就会很简单，所以要让集成点一下按钮就能快速完成。人们问我应该多久集成一次……

告诉你的开发者每天至少集成一次。

实际上，这样做有些问题。通常最后一个留下来集成的人会面对一堆测试失败而没法回家吃晚饭了。而这将是他最后一次这么做了。

实际上，应该保持**随时**集成，一个小时左右，甚至更频繁，每添加了一个小小的功能就集成一次。集成一小点并不困难，很容易找出并修复问题，因为通常只需要修复一两个小地方就能让测试全部通过。如果测试失败很可能是新引入的那一小点造成的。

这真的很简单，而且通常是自动的。如果我输入了一些错误的东西，就没法通过编译。编译器告诉我去修改，我就改了。然后进行编译，通过测试，自动地把代码提交到构建系统中。

甚至有些自动化的测试工具可以在你每次按键时进行构建。你甚至不用点击**构建**按钮。你只需要编码，代码被编译（如果你输入的代码是可以编译的）后将进行自动构建。就是这样，一切都是在后台完成的。

本书后面讲到测试驱动开发（Test Driven Development，TDD）的时候会更深入地讨论这方面的内容。不仅仅是你的代码会自动编译，所有的测试也是自动执行的。你会看到一些绿色的进度条，告诉你"所有的测试都通过了"。这对我来说就像一杯浓咖啡一样。让我保持清醒。让我精力充沛。

7.6　迈出第一步

优化软件开发首要的因素是构建自动化。软件产品，尤其是那些通过光盘或者其他介质发布的产品，有着发布和后续维护相关的成本。无论你决定何时发布，软件应该从第一天起就**具备发**

布条件，而且在开发阶段需要一直保持这样。

对我来说，实施敏捷和Scrum并不仅仅意味着“迭代”或“站立式会议”，而与你集成的方式息息相关。如果你实施双周迭代，但每个团队都把他们的代码集成到自己的分支，然后在年末把所有团队的分支集成到一起，那么我告诉你一个坏消息——**你正处在瀑布模型之中！**

如果你的软件有99%的完成度，最后的1%中可能包含着未知数量的风险。相反，应该在构建中把功能全面集成进系统。

一个健康的构建版本是一个健康项目的核心。消除所有的分支，利用**功能标识**来控制正在开发但尚未完成的功能是否需要构建，保持持续集成。通过**模拟**组件消除不必要的依赖，加入自动化单元测试。如果你这么做，你就成功了一半——正如谚语所说：开始是成功的一半。

我发现恐惧最能阻止人们前进。一旦你克服了恐惧，发现了究竟为什么这些流程会起作用，那时就不仅不再令人生畏而且会令人兴奋了。你可能担心这些实践会让你的生活变得更复杂，实际上，在短期内确实会有一点。

停止在发布周期的末期，每个人都和爱人说拜拜，花上两周时间试图让无法工作的软件正常运行……破除这些旧习惯，取而代之的是准时回家吃饭，也许刚开始有些困难，但是很快就会从“困难”变成“值得”再到“我之前怎么会那样行事”。

这就是**龟兔赛跑**的故事，缓慢且稳定，会在比赛中获胜。无论我们的角色是什么，都要有大局观。对于新事物都会有认知负荷，但是如果想让效率、可扩展性和可维护性获得几何式增长，我们就必须提前做些功课——得承认目前的做事方式有问题。

我们必须共同想办法解决。

7.7 付诸实践

以下是把这些想法付诸实践的方式。

7.7.1 构建敏捷设施的 7 个策略

实现敏捷和技术卓越的第一步是建立相关的基础设施。自动化的构建服务器至关重要，因为一个用户故事只有在被完整地集成到构建版本中的时候才能称其为“完成”。下面是构建敏捷开发良好设施的7个策略。

用版本库管理一切

在过去二十年里我还没有见过不使用版本库的开发团队。版本库对于所有的开发过程——敏捷、瀑布模型或者其他——都是核心的工具。但我见过许多客户都没有把对构建来说至关重要的非源码文件（配置文件、脚本、存储过程等）进行版本管理，难怪他们的发布版本很不稳定。修改起来很简单：把构建所依赖的一切事物都进行版本管理！

一次点击全部构建

将整个构建流程全都自动化，这样，代码在本地保存后会被编译，自动执行测试，如果所有的测试都通过，自动签入到版本库，在服务器上进行构建，执行更多的测试——所有的一切都在几秒内完成。

持续集成

持续集成并非意味着每次迭代都要进行发布，但是**如果你想发布的话就可以！**持续集成是敏捷的关键。大多数工具都是免费的，而且从中得到的反馈非常有价值。

为任务定义验收标准

每个任务都要有定义良好的验收标准，好让你知道什么时候算完成。有一些自动化验收测试框架（比如SpecFlow、FIT或者Cucumber）可以帮助你。验收标准不仅帮助你明确什么时候任务算完成，而且帮助你避免过度开发。

编写可测试的代码

一旦一个团队进行自动化测试，生活就会变得容易许多，这并不仅仅是对质量保证人员（他们不再需要浪费时间在回归测试上）来说的，而且也是对开发者来说的，他们可以立刻得到关于工作的反馈。编写自动化测试还有另外的好处：你逐渐开始编写更容易测试的代码，这样的代码本质上会比难以测试的代码质量更好。

保证必要的测试覆盖率

作为一个完美主义者，我对我写的代码追求百分之百的测试覆盖率，即使我知道这并不总是能达成。因为在编写代码之前编写测试，更容易达到较高的代码覆盖率。但是，对于没有提前编写测试的开发者，如果团队对测试覆盖率有强制要求，他们有时会对更容易测试的代码（比如 `getter` 和 `setter` 方法）编写测试，而让真正需要自动测试的部分没有被测试覆盖，从而引发更大的问题。

即时修复失败的构建

能工作的构建版本如同项目的心跳一般，当构建失败时整个项目戛然而止。永远不要让这样的事情发生。每个提交代码的人都需要对他们的代码负责，保证其正常运行。如果提交的代码造成了构建失败，必须**立即**修复或者回滚。

成功的自动化构建是了解项目情况的关键。集成通常是瀑布模型项目中最糟糕的阶段，让真正的问题和进展的真实情况难以评估。所以与其把集成放在项目的最后阶段，不如每天都进行一点集成，直到一个功能在完整的系统环境中运行，这样才算真正完成。

7.7.2 消除风险的 7 个策略

软件开发是充满风险而且昂贵的。软件没有实体而且难以理解。看似无关的组件之间有着细微的联系。优秀的软件开发流程着眼于通过在问题演变成致命缺陷前尽早发现而降低风险，此时更容易解决。

持续集成

创建可以从第一天就进行构建的系统，并在开发阶段对软件进行持续集成，这是消除风险的唯一方法。将集成推迟到临近发布的最后阶段是个糟糕的主意，因为只有在集成的时候才能看到我们的代码和系统的其他代码是否兼容。最难缠的bug通常在集成阶段发现。一个功能在集成到系统之前都是未经检验的，而且不知道包含多少未知风险。发布之前将大量的功能一起集成如同在拉斯维加斯赌博一般。你的胜算不大。

避免分支

代码集成到系统之后风险降到了最低，但是，如果组件是通过分支进行开发的，在分支被集成之前，发布前集成同样也是风险未知的。不要用分支，要使用“功能标识”来控制系统中尚未完成的功能是否需要构建。

在自动化测试上下功夫

移除待发布版本验证上的全部人工干预，完全进行自动化处理，这可以从根源上降低开发成本。快速的自动化测试让你可以在任何时刻进行测试，得到重要的反馈。如果发现你的系统难以进行自动化测试，那么考虑重新设计，好让某些部分更容易测试。缺乏可测试性的系统往往意味着糟糕的设计。

识别风险区域

风险通常和未知的事物或不在我们直接控制范围内的事物相关。通过研究哪些事情会出问题将其甄别出来。识别出外部依赖，即那些不在你直接控制范围内的事物，然后想办法降低风险并隔离依赖。

征服未知

一旦未知被识别出来，你就可以在短时间内对其进行处理，然后将其签入以保证进度。刺探（spike）通常用来处理一个或多个未解决的问题。通过“时间盒子”和“即时签入”来控制进度，避免陷入无休止的耗子洞中浪费时间。尽量保持对未知的隔离，让未知变得越来越小。

构建可以体现价值的最小部分

小的问题更容易理解、解决、验证和维护。但是要把问题分解到多小？我的首要原则是构建最小的可体现价值的部件。如果80%的价值来自于20%的功能，那么先构建这20%。我们也许根本用不到剩下的80%。

频繁验证

我们的客户也许在见到软件之前都不知道他们想要什么。尽早进行验证可以帮助我们构建出更高价值的产品，让客户参与其中找到更好的解决方案。当开发过程中客户或产品负责人和开发者形成了伙伴关系后，我们通常可以比“提前计划好一切”更能构建出优秀的功能。

降低软件中的风险主要是靠确保构建方向和方式的正确性。如果可以尽早并且频繁地从用户那里得到反馈，那么就可以知道我们的方向是正确的。遵循优秀的工程实践帮助我们编写出容易修改的代码，并持续地进行集成，我们可以确保构建的方式是正确的。做好这两件事情可以显著提高你的成功率和效益。

7.8　总结

持续地进行集成，因为一个用户故事直到“完美完成”之前都不算是完成。我们的目标是尽快从头到尾完成一个用户故事，这需要自动化构建——健康项目的心跳。

本章中心思想如下。

- 在构建期间对代码进行集成可以降低软件开发中的风险。
- 集成很让人痛苦，所以在瀑布型中把它放在后期，这增加了修改代码带来的风险和成本。
- 对发布版本进行自动测试，让最后期限前的修改成本可以忽略不计。
- 更短的反馈回路让开发者可以立即看到他们的所作所为造成的结果。
- 了解到持续部署有多重要之后，你会开始寻找让各项任务自动化的方式，并且用持续集成迅速得到新功能在系统中运行情况的即时反馈。

通过在代码编写阶段就进行集成，我们可以降低软件开发的相关风险，开发者可以从中得到充足的反馈。这降低了发布前紧急修改的成本，缩短了反馈周期，确保系统时刻处于可发布状态。

第 8 章

实践4：协作

我们拥有的最宝贵的资源是彼此。

齐心协力、互相学习、共同进步的时候，我们可以征服一切。但协作不是与生俱来的。和其他事情一样，想要成功协作，必须付出相应的努力。

没有人（但愿真的没有人）第一次坐到方向盘前面就参加驾照考试。我们要先拿到新手许可，学习交通规则，在空旷的停车场进行练习……

先学习开车，**再**去考驾照。

每个技能背后都有方法论、技巧、实践和原则，协作也是一样。

软件开发者是信息工作者，而信息工作者的工作依赖于信息。

这与那些工业革命时代留下的想法形成鲜明对比。大家都被教导按照一定方式做事，对工作场所有着特定的预期。我们被教导着希望有自己的办公室，或者至少有自己的办公桌。我在IBM工作的时候，这样的环境是所有资深开发者所期待的：一间属于自己的可以关上门的办公室，漂亮的家具和足够的私密性。如果你**足够**优秀的话，甚至会有属于自己的窗子！

但事实上，这是一个很大的误区。

儿童心理学家发现了**平行游戏**的概念①。如果将婴儿放到一个公共区域内，给他们一组玩具，每个孩子都会拿起一两件玩具开始玩耍，但是无论家长怎么尝试让他们一起玩，他们就是不肯听。每个孩子都拿着各自的玩具自娱自乐，好像对周围其他孩子毫无察觉。

好像毫无察觉——但他们并不是真的对其他孩子毫无察觉。他们都一边盯着自己心爱的玩具，一边看着周围的其他孩子。他们互相观察，看其他拥有同样玩具的孩子如何玩耍，学习其他孩子的行为，以及什么看上去有意思，等等。但他们并不直接交流。

① What to Expect. “What’s Parallel Play?” Accessed November 28, 2014. http://www.whattoexpect.com/playroom/playtime-tips/what-is-parallel-play.aspx

就好像这些孩子坐在他们自己的办公室里，隔三岔五地走出去看看其他孩子在做什么一样。

但到了三四岁的时候，他们就开始和其他孩子交流了，和其他孩子分享玩具，或者，实话实说，从其他孩子那里**偷取**玩具，并且开始尝试着探索复杂微妙的人际关系，直至中年这都会是个挑战。

假设本书读者都至少四周岁了，仅仅身处团队之中是远远不够的，你需要实实在在地成为团队中的一分子，完全**融入**团队文化之中。

我在IBM做合同工的时候，合同工“享有”的是二等公民待遇。我们没有自己的办公室。我们被安置在被称为“牛棚”的地方，一个所有桌子都摆到一起的大房间，每个人都面对面工作。这个名字有些贬义，把我们当成了牲口。我也听说过称呼类似陈设为“矿井”或“西伯利亚”。

他们无法理解为什么我们可以比那些领着高薪的IBM雇员更有产出。

我们的效率之所以更高，很大一部分原因是我们可以协作。我可以抬头看到我的同事，提出问题，回答问题，讨论问题。

8.1　极限编程

极限编程的核心思想来自于Kent Beck和他在此领域内的早期作品。Kent Beck在《解析极限编程：拥抱变化》[Bec00]中创造了“极限编程”这个词，这本书描述了他和同事Ward Cunningham以及Ron Jeffries在“克莱斯勒综合薪酬系统”的工作中开发出来的一套方法论。这个系统庞大且昂贵，而且面临着失败。为了让其回到正轨，克莱斯勒邀请Kent Beck作为顾问来看看他们的情况，针对大家公认的缺陷给出修改的意见。

Kent和团队进行了交流，其中每个人都知道项目出了问题，CIO建议Kent来带领团队，但他坚持作为顾问。Kent带给团队的一项改变就是新的办公室布局，如下图所示。隔断被公共办公桌所替代，设置了更多的公用设备，消除了“私人”空间。



这对项目和工作环境来说都是一个“转折点”，一年之后团队的境遇完全不同了。以前人人自危的开发者开始紧密合作来挽救整个项目。改变之一就是团队成员将工作任务视为比办公室空间重要。尽管看上去是表面上的改动，仅仅是重新布置了陈设，但这种开放性的思想不能被轻视或忽略。

8.2 沟通与协作

软件开发是一项社会性活动，需要相当多的沟通和交互，需要不间断地学习和交流，对抽象事物进行处理和讨论，所以不同个体之间的协同极其重要。管理者需要牢记，一线开发者最清楚情况，而没有投身于软件开发中的人往往无法真正理解软件开发。

对于有些人来说这也许是个倒退，他们将公共空间视为对隐私和个人空间的侵犯，而对于我们这样“特立独行”的人来说却不是这样：如果你在监狱里卷入了一次斗殴或者其他不端行为，你会被关禁闭。自己一个人处在一个无窗的小房间里是一种**惩罚**。这怎么成了美国大公司里面的“特权”？

但是同样地，为了能够让合作环境发挥最大作用，还需要学习许多技巧，采用各种实践，仅仅构建一个开放的空间是远远不够的。我们在开放空间里的行为，即**彼此**的交互，才是公共空间的目的，也最能发挥出公共空间的影响力。

软件开发不仅仅是一项技术活动，它同样也是一项社会活动。团队成员必须能够对复杂抽象的问题进行沟通并良好地协同工作。沟通依赖于对知识的共同理解，而不仅仅是对空间的共享。所以我们必须统一对目标的定义，对质量的定义，以及对“完成”的定义。我们必须用设计模式、重构和其他通用实践这些统一的语言，而且必须互相帮助结对工作。

有个常见的误解是软件开发者都不善交流。我们实际上是出色的交流者。我们也许不擅长闲谈，因为我们喜欢有营养的交流。正如我所说的，我们实际上是喜欢所从事的工作，喜欢开发软件，喜欢对软件开发有热情的人。软件开发是世界上最具合作性的活动之一。

8.3　结对编程

极限编程中最有价值同时也是最被轻视和误解的一个实践就是**结对编程**，结对编程是两个开发者在一台计算机前执行同一项任务。

经理们跟我说，他们不希望开发者进行结对编程，因为没法承受失去一半的“资源”——但结对编程并非轮流使用计算机，而是让两个头脑来解决同一个问题，这样会比两人各自单独工作更迅速并且质量更好。

结对编程是极限编程中最难让开发者尝试的实践之一，但是若使用得当，这种方法最有效。我们常常认为程序员喜欢独自工作，认为编程是项独自进行的工作，但是软件开发者一起工作的时候会比单独工作更高效。

你是否尝试过自己一个人搬家？

你需要自己搬运每一件家具，独自把家具搬上卡车，没人帮忙卸车，完全靠自己把家具搬进新家。如果没有大件家具还好说，也就是多花点时间。但是如果你有任何一个人搬不动的大件家具的话……

哪种方式更快且效果更好：自己一个人建造整座房子，还是有一队专业木匠、水管工、电工一起用他们各自的专业工具来建造？

像搬家具和盖房子这样的体力活是明显的例子——一个人无论多么强壮，体力都是有限的。事实上，人的脑力也是有限的。抽象的概念性问题可能跟超级大床垫一样沉重，而有人在脑力上“助你一臂之力”对软件开发者来说和建筑工人一样有效。

尽管如此，还有许多开发者对结对编程抱有抵触心理，甚至拿这个概念开玩笑，就是不愿意尝试。

但是一旦真正采用了结对编程，我保证那种体验超乎想象。当然，正确的做法至关重要。我见过有人尝试结对编程并且认为这种想法有缺陷，但其实是他们的做法不对。他们就像两个坐在一起的婴儿，偶尔交换一下玩具，自娱自乐，而不是一起玩耍。

8.3.1 结对的好处

结对编程能帮助知识在团队中迅速传播，比我知道的任何其他方法都要有效，而且对于复合型的团队来说，所有成员都熟悉整个代码库很重要。结对编程可以防止团队成员过于专门化，并且帮助团队达成共识。

结对编程对于命名这样的工作尤为有效，这件听上去微不足道的工作在软件开发中却是非常重要的。几乎软件中的每一件事物都需要命名，而名字揭示了其背后的意图。用“见名知意”的方式给各种事物命名对于编写优秀的软件非常重要。

在处理复杂问题时，近距离的合作会帮助你验证自己的想法，有助于对问题进行更全面的思考。

结对编程可以帮助彼此提高速度，有助于资深开发者指导那些经验尚欠的开发者。

遵循优秀的开发实践可以让开发者互相学习、互相支持。

在有人旁观时我们更不容易偷懒或者编写出糟糕的代码，所以结对编程可以产出更容易维护的代码。对于设计、调试、重构等任务，结对工作也同样有帮助。

结对编程有助于形成统一代码风格和代码集体所有权。

代码集体所有权对于软件开发非常重要。在传统的瀑布模型环境中，每个人都有自己的一组代码，工作风格各不相同，短期影响是代码难以阅读，长期影响是代码难以维护。

我并不关心具体应用哪一种风格，关键是在团队中要保持统一。

理想情况下，你应该没法通过阅读代码来看出是谁编写的。我不赞成用标准文档来定义编码风格。编码风格应该用代码本身来体现。

统一的实践，统一的代码风格，意味着你必须帮助团队中的新成员上手。为达成这一目标，没有比结对编程更好、更有效的方法了。

让团队新成员和资深开发者（或者至少是有经验的开发者）一起结对一两周。在那段时间里更像是“见习”而非结对编程，资深开发者更多地扮演了导师的角色，而真正理想的结对编程关系则是基于平等的合作关系。餐馆中的服务员就是这么做的，而他们并未肩负百万美元级别的软件项目。

根据Alistair Cockburn和Laurie Williams在他们的论文《结对编程的成本和收益》中所述，当两个程序员一起工作的时候，并不是两个人做一个人的工作，这会将团队效率降低50%。事实上，结对编程节省了15%的总编程时间。[①]但是，即使结对编程消耗了一点额外的时间，这些时间也

① Cockburn, Alistair. Williams, Laurie. “The Costs and Benefits of Pair Programming.” Proceedings of the First International Conference on Extreme Programming and Flexible Processes in Software Engineering (XP2000). http://dsc.ufcg.edu.br/~jacques/cursos/map/recursos/XPSardinia.pdf

可以在更快地解决复杂问题的时候得到弥补。

结对编程编写的代码中缺陷的数量比单独编写的代码中的要少得多。结对编程也可以用更少的代码解决同样的问题。需要处理的代码更少，并且那些代码更优质、更高效，这显然会大幅度降低维护成本。而软件的维护成本是非常非常高的。

开发者在结对编程中很少会被打扰，事实上是**极少**被打扰。人们不太会去打扰在一起工作的两个人。当你看到两个人一起工作的时候，你不太可能会走过去说“你们俩都要停下手中的工作，我要带走一个人跟我去开会”，或者“我要问你们其中一个人一些问题，还有你们都看了测试报告的摘要了吗”，或者“嘿，海鹰队的表现怎么样”。

但是当一个人在自己的隔断里面噼里啪啦打字的时候，人们更可能会走过去说：“你能在这上面签个字吗，你能给那个做下初始化吗？”有些事情也许是必须做的，比如整个团队可能都在等着你起草某些文档。但是打断一个即使仅有两个人组成的队伍所带来的压力，也会让人们更容易分辨出哪些事情是紧急的，而哪些可以等上一两个小时。

而且并非只有外部的干扰让人分心。在旁边有人等着一起完成任务的时候，我们不太会去回复邮件，刷刷微博，或者开其他的小差。任何人都有犯迷糊的时候——而开发软件需要很多的思考——从坐在你身边的人那里寻求一些新思路，或者咨询些问题，能帮助你更好地集中注意力。

在结束一天的结对编程后，我发现自己精疲力尽。我完全没力气了，但我感到十分满足，因为我学到了很多东西，而且完成了很多任务。当人们结对的时候，他们互相督促，每个人都充实地完成了一天的工作。

各位经理，这就是应鼓励你的开发者结对编程的原因。

8.3.2　如何结对编程

做任何事情的方式都不止一种，也总会有一些方法比较高效。

把两个人放在一台计算机前，然后说“好了，开始结对编程吧。我会在三个小时后回来检查”，这么做显然是不够的。当然，开发者知道如何编程，也知道如何彼此交流，但他们并不一定知道如何让两个人一起高效工作。

正如名字中体现的，结对编程通常需要两个人在一台机器上工作：**驾驶员**和**领航员**。关于结对编程的一本好书是《结对编程解析》[WK02]。

驾驶员是键盘前的那个人，领航员坐在他旁边能够清楚地看到整个显示器。

但是领航员并不是干坐在那里看着驾驶员编码。领航员和驾驶员彼此交流。这应该是真正的交流，不仅仅是坐在旁边对你进行批评，指出你的错误。对你搭档的成就进行认可和说“哎呀，有个拼写错误”一样重要。更重要的是要提出“为什么你这么做而不那样做”，然后听他解释。

答案可能会让你大吃一惊。

参与，讨论，交流。

然后转换角色。

驾驶员交出键盘鼠标成为领航员，领航员变成驾驶员。

驾驶员和领航员应该尽可能频繁地交换角色——多于五分钟但是不能多过三十分钟。我习惯每二十分钟进行交换，但有时候更加频繁。交换应该在适当的时候进行，而不是用计时器，尽管计时器在一开始可以作为后备方案。两个有着共同目标和基本沟通能力的人应该可以分辨出那些自然的停顿点，事实上大部分的开发者确实能做到。如果驾驶员觉得领航员有些无聊，那么就要更频繁地交换键盘。有时候两列思维的列车其中一辆比另一辆更快，让慢一点的那个人控制键盘，我保证可以让他立刻变快。

我觉得非常有用的另一个技巧是乒乓结对：一个人编写测试，另一个人保证测试通过，清理代码，编写下一个测试，然后把键盘交给第一个人保证测试通过，清理代码，编写下一个测试……周而复始。如果我们能在五到二十分钟内让测试通过，乒乓结对非常有用。但如果我们在某些地方卡住了，就要回到每二十分钟切换驾驶员和领航员角色的方法了。

8.3.3 和谁结对

谁应该和谁来结对是一个逻辑问题。有三种不同的方式来进行配对，每种都有其优缺点。

第一种方式，我们可以根据开发者的强项和缺点来配对，他们可以发挥各自的特长，并且帮助克服彼此的缺点，包括性格上的。当通过性格来分组的时候，把主动型性格的开发者（外向，有主见）和被动型性格的开发者（内向，沉默）匹配到一起。我们要的是打破惯性思维，让两种不同的头脑在一起工作。反之，让两个好朋友（或者性格类似的人）在一起组队却不是什么好主意。避免让两个人仅仅是“随声附和”。

第二种方式，让最有经验的开发者和经验最少的结对。在团队加入新成员的时候这是最有效的方式。

如果项目有着严格的时间限制，或者因协同不佳而难以施展，或者团队有新成员加入，我通常都会把最有经验的人和经验最少的人配在一起。当然，这意味着把资深开发者放到导师的角色，资深开发者在指导学生的过程中也会学到更多的东西，往往总是如此。一生中大部分时间身为一名教师，我可以负责任地告诉你，即使是面对最熟悉的领域，真正能扩展思维的最佳学习方式，就是去教授它。对其他人进行讲解，当你听到自己说了些什么的时候，你会对自己大吃一惊。

正如普鲁塔克说过的：“思维不是等待承载的器皿，而是等待点燃的火炬。”[①]教学不是把我

① http://en.wikiversity.org/wiki/Plutarch_quote

头脑里的东西灌输到你的头脑。教学是一个共同探索的过程，是让学生展现思考的过程，让他们自己发现答案。他们问的每一个问题都是一个帮助他们学习的机会，所以我通常都不会直接给出答案。我会试图帮助他们自己想出答案，因为思考过程是最具启发性的，也是最有价值的，对于一个问题往往没有统一的标准答案。

第三种方式，也是我十分推荐的，是**随机**配对各位开发者。一次次的随机配对，让我们可以和所有人紧密合作进而更好地发挥，这样并不是忽视了人们编程方式的不同，而正是因为编程方式不同，才需要结对来磨合。我们很难从一个和你的想法完全一样的人那里学到新东西。

我坐下来和那个角落里的少言寡语的伙计一起工作的效率是和那个我认为的最佳搭档一起工作时的三倍。在和所有人都搭档过之前你没办法预先知道这件事。结对一天，一个小时，或者一个星期，然后进行轮换。

另外一个我认为很有效的结对方法是Llewellyn Falco倡导的**强制性结对**[①]，遵循如下规则：

> 一个想法在由你的脑海输入到计算机之前，**必须**要经过其他人的手。

你在说话的时候和在打字的时候，大脑的活动区域是不一样的。对某些事情进行描述常常有助于让你对之前没想到的细节产生更清晰的认识。这就是为什么结对编程的两个人都需要保持活跃的状态。

一份关于结对编程的优秀论文是Arlo Belshee的《随意配对以及初学者的心态：拥抱缺乏经验》[②]。他在论文中探讨了一组流程以及该流程如何帮助我们结对编程：

> 流式的思维状态是最需要关注的。整个问题和解决方案空间都被装载到了开发者的大脑中。开发者在流式工作环境下的效果要好上一个数量级。
>
> 流式的结对也是一样。解决方案和问题空间被两个参与者的大脑共享。流式的结对也比非流式的结对效果好得多。

我们在此处讨论的是针对软件开发者的结对，但我认为任何团队都能从结对中发现价值。

8.4　伙伴编程

有些团队对于结对编程有着严重的恐惧，而且这样的团队不在少数。在此，我建议使用另外一个实践。我称其为**伙伴编程**。

有时候虽然独自工作比较好，但是开发者依然希望得到反馈。进行伙伴编程的时候，在一天的大部分时间里你依然独自工作，和平时一样。在每天最后的一个小时里，你和你的伙伴对一天

① http://llewellynfalco.blogspot.com/2014/06/llewellyns-strong-style-pairing.html

② Belshee, Arlo. “Promiscuous Pairing and Beginner’s Mind: Embrace Experience.” http://csis.pace.edu/~grossman/dcs/XR4-PromiscuousPairing.pdf

的工作进行代码审查。这样做没有结对编程那么极端，但是一样可以得到很大的好处。

和结对编程一样，我建议每天轮换伙伴，或者至少每周轮换。也可以按每个任务、每次迭代等不同的方式轮换。在某种意义上说，伙伴编程是结对编程的前身。有时候你必须让人们去尝试一些新事物，而伙伴编程是结对编程的一个安全尝试方式。

软件开发因人而异，而且成败难料。在他人面前工作对有些人来说很不舒服。应该给人们一些适应新实践的机会。

亲身体验是非常重要的，上面说了这么多，只有你试过才知道。纸上得来和亲身体验有很大差别。

我曾不止一次见过，在尝试过结对编程几天之后，团队里曾经最抵触的人变成了结对编程最大的支持者。

8.5 穿刺，群战，围攻

除了结对编程和伙伴编程，还有一些其他的合作开发方式值得一试。

8.5.1 穿刺

穿刺是指两个或两个以上开发者一起执行同一个任务，通常用预先规定好的一段时间对某些未知问题进行研究。穿刺对于这类场景十分有效。我们有些未知问题，而且必须想办法处理，所以进行一次“穿刺”然后做些研究。就像为一个短期问题组建一个委员会，一旦问题解决就解散委员会。

8.5.2 群战

群战是以整个团队（或两个以上开发者组成小组）对同一问题进行处理，但是他们是同时进行的。这对于那些重大问题来说是个好办法。如果团队被某些大问题困住了，那么整个团队都去想办法解决它。这将十分有效。

8.5.3 围攻

围攻[1]是由Woody Zuill和他的团队发明的概念。整个团队针对同一用户故事进行工作，像一群蚂蚁合作拆分一大块食物一样。

虽然看上去十分低效，但事实证明针对特定类型的项目却十分有效。

---

① Zuill, Woody. Blog: Mob Programming. http://mobprogramming.org/

团队所面对的是一个非常复杂的项目，所有人坐在一起讨论要如何处理。他们发现一起工作效率非常高，所以那天结束后就决定转天再试试，然后再也没有停止过。

他们现在经常这样全体出动。把办公室布置得和会议室一样，房间里有两台投影仪。每个人轮流操作键盘，其他人（五个到七个）作为领航员。

你可以在他们的网站上看到一部延时摄影视频[①]，把他们团队一天的编程活动在五分钟内展示出来。



在这个视频中你将会看到团队中的每个人始终聚在一起工作。它将合作提升到了一个全新的境界。有的团队发现在尝试新技术的时候围攻格外有效。

8.6 在时间盒子中对未知进行调研

除了培训软件开发者，我还为科研公司提供咨询。有一家客户是大气科学界的权威，研究天气之类。另外一家客户则是对地底下的东西进行研究的，研究地质拓扑和地质制图等。两家公司都有大量的科学家：纯粹的、货真价实的研究型科学家。

有意思的是，他们不仅仅把开发者带到了我的课堂，还将科学家也带来了。我将传授给开发者的技巧同样也教给了那些资深的科学研究者们，他们同样发现这些技巧非常有效。归根到底，软件开发者也是研究者，也是真正的**创造者**，对我们来说被证明是成功的技巧对科学家来说也是

① http://mobprogramming.org/mob-programming-time-lapse-video-a-day-of-mob-programming/

一样。我教给他们穿刺、时间盒子、迭代、测试先行等，深受欢迎。

时间盒子对于研究未知的东西十分有效，目标是对其进行一次穿刺。在开始穿刺之前，最好提前在脑海中准备好一组问题、目标或者目的。简单来说穿刺就是：

> 在这段时间——可以是一个双周迭代，也可以是一小时或者其他时长——我将对**它**进行研究。

假设“这些是我已经知道的，这些是我不知道的”，然后在未知的领域上画一个圈，随着穿刺的进行不断缩小那个圈，直到完全消失，都被**化**解成为了已知。如果你做不到，那么试试看是否能将未知进行封装，这样当你以后学会更多时就能处理。

8.7 定期代码审查和回顾会议

结对编程本质上就是**在你写代码的时候**进行代码审查。事实上，这就是极限编程这一词语的由来。Kent Beck将其称为极限编程，是因为要将那些对我们有用的经验发挥到极致。

坦白来说，我并不认为极限编程有多“极限”。从很多层面来说，软件开发本来就该这么做。一些业内最保守、最有商业头脑的人也都推崇极限编程。

在极限编程的想法中，如果代码检查是好事，那么为什么不在写代码的时候逐行审查呢？这就是结对编程的由来。这就是代码审查的“极限”版本。

但是，即使在写代码的时候互相审查，也无法替代真正的代码审查。我们依然需要让团队中的所有人都理解系统中全部的代码，而不只是结对伙伴才理解。整个团队都需要理解代码的设计和我们所选方法中的取舍。

让团队决定他们多久需要对设计和代码进行审查。每当一个开发者完成了某一功能，他就可以向其他成员讲解他是如何编写的。我个人十分享受代码审查并且从中学到了很多，但是我见过很多的审查变成了对代码格式的争论。设计和代码的审查应该首先并且着重指出**设计思路**并说明为何选择这种设计。理解设计中做了哪些取舍以及其是否方便从不同角度扩展，这才是代码审查中应该讨论的。

回顾会议无论是在迭代之后还是在发布之后都是有益的。回顾会议帮助团队整体进步，在回顾会议当中可以发现真正可做出响应的问题。回顾会议即使只发现了很小的问题也会得到巨大的回报。如果可以对问题做出响应，主动修正它，进而发现导致这个问题的反模式，就可以防止问题再次出现。

回顾会议可以是非正式的。让团队聚在一起花一个小时左右，每个人都有机会发言，说说哪些正常哪些有问题。医生称之为“反思”，军人称之为“简报”，但意思都差不多：我们做了什么，是如何做的，为什么做，下次怎样做得更好，等等。

8.8　加强学习和知识分享

按旧时代的惯例，稳定的工作来自于特殊化。掌握一些让你变得无可替代的知识，会让你免于被裁，甚至升职加薪，而且可能成为公司内部博弈的筹码。那样的话，谁不想把那些知识据为己有不与他人分享呢？

但时至今日，稳定的工作——引申一下，稳定的**职业**——所需要的恰恰相反。我们觉得团队中那些乐于分享知识的人更有价值。结对，群战，围攻……这些都是帮助人们合作的技术，让我们做得更多、更高效、更优质。

组建可以处理各种任务的复合型团队，可以自行设计、实现、测试自己的软件。即使你在一个项目里是前端开发者，在另一个项目里是后端开发者，作为一个专业软件开发者，你需要有一组与你使用的平台和语言无关的原则和实践。

结对编程不仅仅是更快更好地编码——这还不够。结对编程**很有趣**。事实上，工作的感觉越像玩耍，我们越容易选择工作而不是玩耍。如果真的热爱自己的工作，其实永远算不上真正的“劳作”，我不是第一个产生这种想法的人。我来分享一个小秘密……

我最喜欢做的事情之一，就是参加一个会议然后引爆全场。

如果没有专家会议、研讨会、报告会，我就找一个朋友、同事，然后我们结对一个早上。我会学到很多东西——那些在一个正式的会议上永远学不到的东西。我会学到节约我无数小时的键盘快捷键。这经常让作为一个软件开发者的我大吃一惊。有这么多的简便方法没有被记录在任何地方。而当我和其他开发者结对的时候，我经常见到晴天霹雳一样的新简便方法——而我也会抖出几个法宝。有的时候发现我们都在用同样的快捷键，只是命名不同。更多时候就如同见到了秘密社团的成员一般，与此同时做出真正的秘密社团不会做的事情：分享信息。

软件行业不能再像秘密社团一样，更不用说各种各样的秘密社团了，每个社团中又有更多的等级，每个社团中又有不同的小团体……孤立的开发者又有着自己的小团体。

请牢记许多开发者（实际上是大部分的开发者）都会处理一些敏感信息。我并不是说“去把你客户的机密公之于众”。我是指在团队内部共享资源，更高效地为同一项目而工作。专业人士应该了解哪些是保密信息并遵守规范。

8.9　诲人不倦且不耻下问

身为软件开发者意味着需要持续学习。几乎每天都会出现新工具和新技术，要跟上时代是个挑战。Scott Bain是我的良师益友，他曾经说过：“时刻准备着诲人不倦，也时刻准备着不耻下问。”

我十分赞同。

作为导师，我曾指导过数以千计的开发者。再次重申，我可以证明最好的学习方式就是指导

他人。我从我的学生那里学到了很多的东西，而在把我的知识转换成容易理解的片段时我的收获很大。作为软件开发者，我从很多同事和伙伴那里学到了很多新想法和新技能。

在结对编程中，我会一次又一次地产生想要自己一个人孤立的冲动，我觉得我们都有过这种冲动。但是当我战胜了内心那个小小的怀疑声音（“别这么做……不会有好结果的……你知道不会有好结果的……”）时，我的收获都很大。

在与超过8000名软件开发者合作过之后，我感觉到我影响了他们，我也从中获益不浅。我可以学习他们的知识，而我的许多学生也从我这里学到了很多。有人跟我说，当遇到棘手的问题时，他们会问自己：“如果是David，他会怎么办？”然后他们就有了答案。我也一样会用假想中的他们来解决问题。我们并肩作战。

通过在一起和谐地工作，我们可以共同构建出更好的代码，并且更容易处理他人的代码。

8.10 让我们付诸实践

以下是把这些想法付诸实践的方式。

8.10.1 结对编程的 7 个策略

在我教授过的所有敏捷开发实践之中，结对编程是最受管理者和开发者抵制的。管理者常常问我，让两个人一起做同一个任务，如何能比两个人分别执行不同任务来得高效……但事实上绝对如此。在开发者每天编写的代码行数上也许看不到很大提升，但是你会发现，开发者可以用更少的代码完成工作，这同样也会降低维护成本，而且你会看到bug数量大量减少，从而大大加快交付速度。在使用得当的时候，结对编程十分有助于团队中知识传播、技术增长和工作满意度的提高。所有的这一切最终都会降低软件开发的成本。这里有几个高效结对的策略。

尝试一下，你会喜欢的

结对编程的体验和很多人想象中的不一样。在一个互相帮助的氛围中尝试，团队成员可以学习如何正确结对，这样可以避免糟糕的体验，进而阻碍日后再次尝试，同时良好的体验会让团队成员继续执行下去。

驾驶员和领航员都要参与其中

结对编程并不是轮流工作。每个成员都有自己的任务，一起工作，并行工作。在结对开发期间，键盘前的人（驾驶员）和在其背后的人（领航员）都要投入其中。

频繁交换角色

每二十至六十分钟交换一次驾驶员和领航员的角色，这样有助于最大化的合作。交出键盘的同时知识也得到了交流，最终通过和其他团队成员进行结对，让知识在整个团队中得到传播。

充实工作一天

结对需要消耗很多精力。你在一天当中并非每分钟都保持状态。在结对的时候更不容易被其他人打断。在结对编程一天之后，我精疲力尽地回到家中，但我觉得极度满足。我总是会在结对编程中有所收获，无论是从比我经验丰富的人那里学到知识，还是给经验较少的人传授一些知识，又或者是我们一起解决问题。

尝试各种配置

有许多的技巧和方案来有效结对。理解它们可以让结对更有效且更高效。尝试通过用户故事、任务、每小时甚至每二十分钟来随机配对。通常，从未想过要结对的人们之间的结对会更有效。看看哪些管用哪些不管用，但一定要尝试所有的选择。结果也许会让你大吃一惊。

让团队决定细节

和其他敏捷实践一样，结对编程不能由管理者强制实施。团队成员必须要自己发现其中的价值。并不是所有人都适合结对，也不是所有的任务都适合结对。有的人比其他人需要更多的时间独处，但和一般认知相违背的是，软件开发更多的是一个社会性活动，需要大量复杂的沟通。而结对非常有帮助。

跟踪进度

数字比文字更有说服力。结对编程**不会**将生产率减半。如果计算“时间价值比”的话，会发现结对编程对生产率有促进作用。跟踪速度、缺陷、代码质量，它们是生产率的体现，会展示结对编程的价值。

在使用得当的时候，结对编程可以显著提高创造的价值，同时减少缺陷。在我所知道的方法中，没有比结对编程更能在团队中迅速促进知识和技能的传播的了。

8.10.2 高效回顾会议的 7 个策略

抽出时间对整个团队进行反思，找出哪些东西还可以提高，这是十分重要的。定期的回顾会议可以让团队养成习惯，经常对自己的工作进行总结，看看哪里可以提高。这里有几个进行高效回顾会议的策略。

找寻小的改进

很多组织都倾向于进行革命性的改变，或者试图一次做出许多改变。小幅度、小范围的改进执行起来更快捷而且更容易。如果每两周仅做出2%的改进，那么一年之后你就有了50%的改进。小的改进更容易接受，而且容易达成！

责怪流程而不是人

当事情出错的时候通常都不是有意为之的，责怪负责人通常会让事情更糟糕。相反，应该找

到导致错误发生的流程上的缺陷。这样可以消除和错误相关的团队成员的压力，好让他们专注于找到避免相同错误再次发生的方法。

5个为什么

通常，表面上的问题都不是真正的问题：它仅仅是另外一个深层次问题的表象。一个查找问题根源的方法是"5个为什么"。当面对一个问题时，问为什么会发生，或者是什么东西造成了这个问题，然后针对那个答案接着问为什么会发生，直到你问了至少5次。差不多在第4次问为什么的时候，通常都会发现一些之前没察觉到的值得注意的问题。

解决根源问题

一旦找到了根源，真正的问题就可以理解并解决了。解决根源问题通常比解决表象问题来得简单，可以彻底解决问题，而不是治标不治本的权宜之计。

倾听每个人的声音

回顾会议应该让团队中的每个人都参与，不仅仅是让一些主要成员发表意见。要求每个人发表意见，给每个人制定可执行的改进目标。持续改进每个人的职责。

给予支持

给予人们进步所需的支持。让人们看到你是真的想要持续改进，支持他们做出改变。如果人们害怕改变，通常是因为他们觉得得不到支持。鼓励他们并且对主动尝试做出奖赏。

度量进度

仅仅设立改进的目标还不够。必须设立可以度量的目标，让每个人都尽力，然后定期对进度进行度量。这是理论联系实际的方式，让人们觉得真实。当人们可以看到目标进度的时候，他们更能全力以赴。

你的流程可以不完美，但是当发现瑕疵或者改进方式的时候，就应该鼓励和支持。一线员工更容易想到改进方法。当工人被鼓励去"停下生产线"来进行改进的时候，整个流程会变得更好。

8.11　总结

协作建立高保真的沟通，推进团队内的知识分享。让我们一起前行！

本章中心思想如下。

- 正确使用技巧来建立高保真的沟通，推进团队内的知识分享。
- 将一些协作技术工具化，包括结对、穿刺、群战、围攻。
- 协作技巧有助于探索未知、加强学习、分享知识。
- 通过代码审查和回顾会议收集反馈并做出响应。

- ❑ 诲人不倦同时不耻下问，我们自己和团队的技能都会得到提高。

我们最宝贵的资源就是彼此。为了最大化协作效果，掌握一些基本的协作技巧和配置十分有帮助。除了结对编程之外，还有穿刺、群战、围攻、伙伴编程。通过加强学习、分享知识，可以改进我们的团队乃至整个产业。

第 9 章 实践5：编写整洁的代码

整洁的代码由Bob Martin大叔在《代码整洁之道：敏捷软件开发技巧》[①]中提出，接着Miško Hevery发表了“代码整洁演讲”[②]。两者都是关于如何编写干净、可测试软件的出色资料。

本章将展示五种代码特质，它们是优秀软件的基础，是软件开发原则和实践的核心。不仅如此，正如我们将要看到的，这些特质和可测试性有着非常紧密的联系。

在理解这几种代码特质之后，我们可以从中推断出几乎全部的优秀软件开发技巧。这五种特质在代码之外的其他地方也随处可见：优秀的小说，引人入胜的电影，电视机的遥控器。具备这些特质的事物更容易理解、更清晰、更直观。缺乏这些特质的事物则看上去混乱又复杂。这是因为我们大脑的结构让我们更容易理解具备这些特质的事物。

商品和服务的特质只能**定性**地描述和形容，而我们将讨论的代码特质则不同，是可以被**量化**的，而且可以被量化得很精确。

代码质量虽是小事但影响很大。对象应该具有定义良好的属性、专一的职责、隐藏的实现。它应该控制自己的状态，而且只应该被定义一次。

我们这么称呼这些特质：

- Cohesive 内聚
- Loosely Coupled 松散耦合
- Encapsulated 封装
- Assertive 自主
- Nonredundant 没有冗余

首字母拼在一起就是 CLEAN（整洁）。

优秀的代码就是 CLEAN 的代码。让我们来看看这些代码特质的细节。

① Martin, Robert C. *Clean Code: A Handbook of Agile Software Craftsmanship*. Upper Saddle River, NJ: Prentice Hall, 2008.

② Hevery, Miško. Google Tech Talks. “Clean Code Talks.” 2008. http://misko.hevery.com/2008/12/08/clean-code-talks-inheritance-polymorphism-testing/

9.1 高质量的代码是内聚的

首先，也是最重要的是，高质量的代码应该**内聚**——每个片段都**只关注一件事情**。这是我最喜欢的代码特质，因为内聚的代码对大多数人来说更好理解和处理。内聚的代码也更容易编写。

如果在英语词典中查找“cohesion”（内聚）这个词，你会发现它有近义词“adhesion”（附着），即描述事物如何纠结在一起。但是软件开发者的“内聚”意味着软件实体（类和方法）应该具有**单一的职责**。

Al Shalloway说过：“不要有**上帝对象！**”

我问Al：“你所谓的上帝对象是因为它试图处理所有的事情，对吗？”他回答说：“不是，我称之为上帝对象，是因为在试图修改它的时候会惊呼‘我的上帝啊’。”

Scott Bain告诉我：“让实体保持一心一意。”

换句话说，一个类应该只有一个目的。

这意味着我们的程序应该由许许多多功能有限的小小的类组成。这是真的。我在第一次读到的时候并不理解内聚的价值。

我是在20世纪90年代初首次接触到面向对象编程的。当时我看到了一个面向对象的程序，而我认为编写它的那个开发者疯了。一个简单的程序用了将近五十个类，而我认为他们之所以那么做，是为了让其他人永远没办法理解，所以他们的工作就成了铁饭碗。

我当时认为，**他们用四十八个类完成的任务我能用三个类来完成**，就好像用的类越少程序质量越高一样。但我并不用这样的方式评判文学。我并不认为一本书比另一本书用的词语少就更好，或者一首诗比另一首短所以更优美。表达应该简练但同时也要**完整**。

采用许多微型类意味着每个类都有单一的职责且做得很好。很明显每个类都有唯一且特有的目的。当我们需要进行修改的时候，可以更容易地专注于一个或几个类，让修改局部化且容易实现。

我觉得一开始难以理解，是因为作为一个结构化程序员，我理解代码的“安全毛毯”是可以跟踪它。我把我当作CPU，然后跟着指令指针在代码中前行。但这不是一个阅读面向对象代码的好方法。

我得知，优秀的面向对象程序像食人妖、洋葱、芭菲[①]（人人都爱芭菲）一样，它们是一层一层的。每一层都代表着不同层次的抽象。这是我们思考的方式，概念一层包裹着一层。这帮助我们将概念“分块”，好让我们可以从高层次处理问题——当我们想知道更多细节的时候，则可以深入到下面一层去。

① 芭菲（parfait）是一种甜食。——译者注

使用得当的话，不同层次的抽象可以帮助我们对组件之间的关系建立起大局观，也让我们可以只在需要的时候关注细节。阅读面向对象的程序和跟踪高度过程化的代码不一样，需要不同的技巧。

如果内聚的类只代表了一种事物，那么这个事物应该是可以被命名的，想法或者概念均可。我们通过语言理解世界，而编写软件归根结底是一种语言性的行为。如果我们可以为某件事物命名，便可以用一个类来代表它。人们本能地会用语言表达概念，所以，如果我们可以容易地为某些东西命名，那么它们往往是定义良好可以理解的。相反，如果我们发现难以为某些东西命名，那么就意味着它的职责还没被定义好。

我说类在只代表一件事情的时候是内聚的，但是如何对复杂的事物建模呢？比如说我们想要建立一个“人类”的类。人是复杂的。他们有交流的行为和走路的行为，吃饭的行为和说话的行为，等等。我们如何在保持类的内聚性的同时，对如此复杂的事物进行建模？

答案是**组合**。

举例来说，一个“人类”的类可以由走路的类、交流的类、吃饭的类等**组合**而成。走路的类又由平衡的类、前进的类等**组合**而成。

我们用类表达不同层次的概念，让它们如我们脑海中的概念一样，一层包裹着一层。

当然，如果系统中的“人类”只需要名字和地址，就没必要再有其他的层次了。但是我在构建一些更复杂的事物的时候，会使用不同层次的抽象。

一个类可以代表一些像独轮车一样的实物，也可以代表一些像银行账户一样的无形事物。一个类也可代表一个想法、一个过程、一种关系，或者任何真实的以及想象中的事物。类定义了**对象**的行为，而对象则在运行时用 new 关键字**实例化**。

很明显，对象不是真的事物。它们是代表，不是它们的名字而是它们的**行为**决定了它们代表什么。名字很重要，因为名字告诉了我们这些类是做什么的，但是在数字世界中，定义事物的不是名字而是**行为**。

9.2 高质量的代码是松散耦合的

这种代码特质是关于保持对象之间关系有意义且清晰的，通常称为“松散耦合”。

松散耦合的代码**不直接依赖**它所使用的代码，这样更容易分离、验证、复用、扩展。松散耦合通常使用间接调用实现。通过一个中间层来调用服务，而非直接调用服务。日后替换服务的时候只会涉及中间层，减少对系统其他部分的影响。松散耦合可以让你在代码间保留一些**接缝**，好让依赖可以被注入进来，而不是直接和它们耦合。

我们通常不直接调用一个服务，而是通过诸如**抽象类**这样的抽象层调用服务，或者在像Java

和C#这样的语言中，可以通过**接口**调用服务。以后可以通过一个用于测试的模拟对象来替换那个服务，或在未来可以增强那个服务，同时保持对系统其他部分的影响最小化。这就是为什么我愿意用抽象而非具体实现来进行耦合。

人们经常讨论**紧**耦合和**松散**耦合，但是我经常忘记哪个是好的哪个是坏的！

紧耦合听上去很糟糕，但是松散耦合也不怎么样。如果一个图形用户界面（graphical user interface，GUI）程序有个“退出”按钮，我不希望它和退出动作是松散耦合的。我不希望它说“今天没心情退出”。我希望在需要的时候进行耦合，在不需要的时候则不进行耦合。为此，我使用Scott Bain在《应急设计：专业软件开发的进化本质》[Bai08]中使用的术语：**有意耦合**和**意外耦合**，这样孰优孰劣就很明显了。

但是系统中糟糕的耦合是从何而来的呢？

真是意外吗？

如果不是小精灵趁我不注意把耦合放在那里，那么这些糟糕的耦合是怎么跑到我的代码里来的呢？

答案是，意外耦合是缺乏优质代码特质的一种体现。举例来说，如果我用一个不够内聚的方法处理各种事情，那么就会有许多无关的原因和它耦合。

这就是为什么要避免编写“全能API”：试图处理过多事情的API是脆弱的、难以维护的，而且耦合了许多毫无关联的类。代码复用有助于消除冗余，但不能以牺牲其他代码的质量为代价。开发者总是以复用的名义编写出“全能API”。

如果API的不同调用者因为不同的原因共有一部分实现或者状态，不要将其暴露。通过创建不同的API来将其隐藏，这样每个调用者只使用他们需要的那一部分。这样做，不仅可以消除冗余，还能提高代码的内聚性，使其封装得更合理，职责更明确，而且可能提高可测试性。

优秀的设计抉择通常不会迫使你牺牲代码质量。我说“通常”是因为有时候出于性能或者其他要求，可能必须做出代码质量的妥协。但是仅当这些要求是真实的而非假设的才需要这么做。正确的设计抉择十之八九会提高代码质量。

在系统存在冗余或者功能分散的时候，耦合尤其是个问题。若同样的问题在系统中各处出现，想要正确地处理问题，必须把系统的各个部分同步。如果在那种情况下忽略耦合，你会得到错误的（或者不准确的）结果。你必须追踪系统里面各种难缠的bug。

9.3 高质量的代码是封装良好的

高质量的代码是**封装良好**的：它隐藏了实现细节。使用面向对象语言而非过程化语言的最大好处之一，就是它可以真正地将实体进行封装。所谓的封装，不仅仅是说让状态和行为变成私有

的。更重要的是，我想用**接口**（做什么）隐藏**实现**（如何做）。

有许多东西可以被封装，编写优质程序的过程，就是在尽可能隐藏的同时完成任务的过程。为此重新定义封装：让“做什么”隐藏“如何做”。你隐藏的实现越多，就有更多的自由修改实现而不影响其他代码。这样代码更加模块化且更好处理。

封装良好的软件是“由外而内”而不是“由内而外”设计的。

由外而内编程：顾名思义，是指从用户的角度出发进行设计。根据客户端的需要设计服务。根据所做的事情命名服务，隐藏它是**如何**工作的。这有助于创建强契约低耦合的服务。

由内而外编程：相反，多数开发者编写软件的方式是将其分解成小块，然后用各个片段组成解决方案。我称其为“由内而外编程”，是因为问题先被分解到能直接解决，开发者可以开始编写代码。但是，如果在理解大局之前直接开始实现，开发者往往会构建出职责不明确的脆弱代码，使其难以封装。

无论如何，开发者需要从以上两个方向去审视他们的代码，但这不是一个顺序问题。如果从细节开始而忽略大局，会让后期的片段组装变得很困难。而从关注大局开始，分析有哪些组件，为什么会有这些组件，就可以更容易地在代码中为其预留出空间。

说到底，软件应理解它所服务的专业领域，清楚该领域在软件中如何体现。领域模型应该让没有计算机背景的该领域专家可以理解。如果编写一个账务系统，领域模型中的对象应该是账户、资产、余额、支票和财务人员熟悉的其他用语。

所以，我希望我的领域模型尽可能地和实现细节分离，这样有助于提高系统的灵活性、一致性、可理解性。从软件如何使用而非软件如何构建出发，进而细化到系统中的每个对象。

当我们观察周围世界的时候，会发现所有事物的可视角度都是有限的。我们在对真实世界的事物建模的时候，也希望去对有限的视角建模。这很重要，因为你能够隐藏多少细节，意味着以后可以在不影响其他代码的前提下进行多少修改。这是真实世界的法则，我们可以在代码中实现概念，也可以利用概念。

封装帮助我们减少修改代码对系统造成的“连锁反应”，而且远远不仅如此。

如果我想知道屋子里每个人带着多少现金，查看每个人的口袋显然不合适。而且，有些人的现金可能不在口袋里放着，他们的现金可能放在钱包（甚至是袜子）里面。我不需要关注每个人把钱放在哪里或是怎么放的。相反，我会问“你们身上带着多少钱”，然后每个人都去查看一下，再告诉我结果。

在用对象来实现系统的时候，让每个对象对自身和所处环境负责，系统中的每个对象都有自己的用途。系统中的每个实体都有自己的职责，如果这些职责改变了，从高层次上看，对系统的其他组成部分来说，这些改变是可以被隐藏的，这就降低了修改带来的成本。

还有许多其他的东西可以被封装，包括关系、过程、差异化行为、过程中的阶段数量、过程中的阶段顺序，等等。可以通过抽象来隐藏概念。方法签名可以隐藏算法的实现细节。通过“适配器模式”或“外观模式”可以用自己的代码封装外部代码。Scott Bain跟我说过：“封装可以将非常差异化的东西让外界看起来有一致性。”

而封装的方式非常非常多：通过方法调用来隐藏实现，通过抽象隐藏思路，通过接口隐藏做事情的方式，等等。

Erich Gamma等四人在《设计模式：可复用面向对象软件的基础》[GHJV95]中列出的所有设计模式都是对不同事物的封装。理解不同的模式封装了什么，对于理解模式并运用模式解决问题非常有帮助。

模式会告诉你：“你不知道的事物对你无害。”

这在生活中并不总是正确的，未知的事物有时候会伤害我们，但在软件中并非如此。你不可能和不知道的东西进行耦合。更少的依赖让代码更容易修改。

默认封装，必要时再暴露。

换句话说，尽可能地隐藏，只在解决问题的时候才暴露。举例来说，先把所有的数据都设置为 `private`，然后在需要暴露的时候提供 `getter/setter` 方法访问 `protected`、`package`、`public` 的数据。一般来说，暴露隐藏的东西比把已经暴露的东西隐藏起来要简单得多。只暴露**解决问题所必需的**，其他的一律隐藏。

当封装成为习惯的时候，你会从**调用者的角度**进行设计，让每个小功能都有可被调用的独立方法，每个方法都有明确的参数和返回值。这不仅仅能通过限制系统内的交互来减少副作用，而且让代码变得简明易懂，让你能准确知道每一个方法需要的参数和它的行为。

也许最基础的封装形式就是用函数签名来隐藏一个行为的实现。这和**设计模式**中的建议不谋而合：“面向接口编程，而不是面向实现编程。”

只暴露解决问题所必需的，其他的一律隐藏。创建优质软件的过程在很大程度上也是尽一切可能封装的过程。谁都能满足一个需求，但优秀的程序员也能封装他们的代码以获得最大的灵活性。

9.4 高质量的代码是自主的

高质量的代码是自主的——它管理自己的职责。软件实体应该是自主的，而不是互相干预。这个代码特质我不经常听人谈起，但它非常有助于判断什么行为该放在何处，并且给于对象应有的职责。

举例来说，控制打印文档的代码应该是“文档类”的一部分还是“打印机类”的一部分？当

我问开发者这个问题的时候，他们立刻回答我说“打印类”应该控制打印文档的代码，但是让我们仔细想想这个问题。当我们讨论谁了解被打印文档的时候，很明显只有“文档类”才知道被打印文档的细节，所以应该由“文档类”来控制打印文档。这并不是说“文档类”需要知道打印机的细节：它将打印任务托管给“打印机类”，但由“文档类”进行控制。

我在加利福尼亚州举办了很多培训，研究出了一个形容代码自主性的特殊方式。在加利福尼亚州很多人热衷自我提升。在西海岸心理咨询是一个繁荣的事业，因为人们希望自我负责、自力更生、做自己的主人。

想象让我们的对象进行“对象心理治疗”。让对象自力更生、自我负责、自我控制。除非有很好的理由，不要让其他对象替它们做它们应该做的事。

首要原则是，对象应该控制自身的状态。换句话说，如果一个对象有一个字段或者属性，那么它同样也应该有管理这个字段或者属性的行为。并不是说这个对象需要自己做所有的工作。以“文档类”来说，它不会去控制打印机里的墨水——那是打印机驱动的职责。“文档类”的职责是表达文档的信息，好让“打印机类”可以对文档进行打印。

这也可以保持对象的良好定义，把行为放到它应该在的地方。当选择哪些对象要拥有哪些行为的时候，观察哪些对象有这种行为所依赖的状态。有时候一个行为依赖于不止一个对象的状态来工作，那样的话，如果没有其他好办法，就选择依赖最多的那个。

对象之间不应该互相干预。它们应该具有**权威**——对自己负责。

不自主的代码就是多管闲事的，它必须经常访问其他对象的状态来做自己的工作。因为它调用了许多其他对象，不如管理自己的状态有效。多管闲事的对象必须访问其他对象的状态才能工作，这打破了封装并且降低了性能。在《重构：改善既有代码的设计》[FBBO99]中，Martin Fowler用“依恋情结”或者“狎昵关系”来称呼那些缺乏自主性的代码坏味道。

如果代码变得过于管闲事而不自主的时候，同一个过程的规则就会被分散到多个对象中，进而让功能变得分散，为了得到正确结果，多个对象就必须保持同步。行为被放置到了不应该在的地方（那些无关的对象）之中，破坏了对象的内聚性，让不同的问题耦合在一起。

这通常是由于在设计中缺失对象导致的，模型中应该出现的对象被忽略或没被发觉。由于识别出了一些必要的行为，它们被放置到了已有的类里面，而没有建立这些行为所属的新类。

自主的特质告诉我们将行为置于何处。如果发现某个方法过分依赖另外一个类的数据，就应该将这个方法置于那个类中。

多管闲事的代码坏味道可能很微妙。类会有许多合作者，所以不认真思考可能很难给行为找到合适的位置。

9.5　高质量的代码是没有冗余的

高质量的软件应该没有冗余：软件不应该自我重复。
高质量的软件应该没有冗余：软件不应该自我重复。

你看，教学上的冗余是有用的。我们通过不断重复来学习新知识。但软件中的冗余往往维护起来代价很大，所以是个负担。

这里我必须谨言慎行。几年前我在达拉斯–沃斯堡教授软件设计课程，当时我不知道有两位资深的美国国家航空航天局的开发者在场。我说："冗余一定是坏事。" 其中一个人站起来说："等等，冗余可以救人性命的！"

他所谓的那种特殊的冗余——有意为之的冗余——是有道理的。如果你曾经开发过"关键任务型"应用，你就知道我说的是什么意思。

"关键任务型"应用是那些一个生产环境中的bug可能让人送命的应用。在这样的环境中思考，和构建社交应用软件时候的思考方式是不同的——如果你有良知而且晚上想要安然入睡的话。开发那样的软件要求不计一切代价的稳定性。这两位NASA的开发者就是构建"关键任务型"应用的。

举例来说，一架航天飞机里有五台电脑，关键性任务需要它们分别计算，然后核对答案。如果答案一致，那么计算极有可能正确；如果答案不一致，就需要重新计算一遍了。

那些NASA的开发者是正确的：对于"关键任务型"应用来说，有意为之的冗余是可以救人一命的。但我这里讨论的是**非刻意性的**冗余——这些冗余**总是**会产生维护性问题。

人们经常说DRY（Don't Repeat Yourself，不要重复你自己）或者"一次且只有一次"，但是找到冗余并非易事。

在代码里的冗余多数很容易辨认。但是，冗余可能以很多不易察觉的形式出现，这就是为什么我使用"冗余"一词而不是极限编程中使用的"重复"。

当我往剪贴板中"复制"代码的时候，我会意识到我正在引入冗余，在"粘贴"之前，我会问自己是否真的需要在两个地方都使用这段代码，还是可以将其放到一个地方然后在这两处进行调用。如果希望在两处都进行调用，可以用一个方法封装它，然后就能通过方法名调用它。

我可以说代码中95%的冗余都是可以轻易识别出来并且去除的。剩下的5%难以察觉，但是通常值得下功夫去处理。我会尽力去查找这些冗余，因为它们会隐藏问题的根源。而不真正理解问题，就没法找到简洁直接的解决方案。如果消除了代码中的每一点冗余，就可以发现之前没有察觉到的问题规律。这样可以把复杂的代码归并成更简单的解决方案。

冗余有许多种形式。我指的不仅仅是状态和行为的冗余，许多其他的事情也可以是冗余的：冗余关系、冗余测试、冗余概念、冗余结构、那些多数步骤相同只有少量不同而让你必须维护两

种不同算法的冗余过程。有一些模式可以帮助我们解决这些问题。

对于我来说，代码中的冗余是试图在不同的地方做相同的事情，无论做的是什么。不一样的代码可能是冗余的，而一样的代码可能是没有冗余的。冗余不仅仅是对形式的重复。冗余是对**意图**的重复。

9.6 让代码特质指导我们

代码特质虽是小事，却可以造成巨大影响。一个对象应该有定义良好的属性、专一的职责、隐藏的实现，它应该自己控制自己的状态、只被定义一次。

这些代码特质指导开发者构建出更优质的软件。

- 高内聚的代码更容易理解和查找 bug，因为每个实体都只处理自身的事务。
- 松散耦合的代码让实体之间的副作用更少，而且更容易测试、复用、扩展。
- 封装良好的代码有助于我们管理复杂度，让调用者不必关心被调用者的实现细节，所以也更容易修改。
- 自主的代码让我们知道行为应该和它所依赖的数据放在一起。
- 无冗余的代码意味着我们可以只在一个地方修复bug和进行更改。

高质量的代码是内聚、松散耦合、封装良好、自主、没有冗余的，或者简称为CLEAN。

值得注意的是，本书中讨论的所有原则和优秀开发实践都有助于提升代码质量。代码质量是衡量软件优劣的标尺，我们讨论的一切都是为提高代码质量服务的。

缺乏这些特质的代码同样也难以测试。如果我需要为一个类编写很多的测试，我就会意识到内聚性不足；如果有很多无关的依赖，那么一定是我的耦合过多了；如果我的测试依赖于实现，我就知道封装出了问题；如果测试结果在被测试对象以外的对象中体现，我的对象很可能不够自主；如果我一遍一遍编写同样的测试，那肯定是出现了冗余。

可测试性成了衡量设计或实现质量的标尺。早些年当我面对同样有效的两种方案的时候，我通常会都实现一遍，然后看哪种效果更好。现在我就简单地衡量一下哪种方式的代码质量会更好，或者哪种方式更容易测试，这样我就能更快找出较好的那种方案。

CLEAN的代码特质互相促进。提升一种特质往往也会提升其他特质，不用同时关注所有的特质，专注其中看上去最有意义的一两种特质即可。

Kent Beck在他简明易读的《测试驱动开发：实战与模式解析》[Bec02]中提到，如果你尽力消除代码中的重复，就会得到高质量的代码，他说得很正确。同样的道理适用于我们前面提到的全部代码特质。它们是事物的不同层面，宝石的不同切面，总的来说就是“优秀的代码”。

当其中的一个层面得到了改善，其他的层面也会跟着改善。当我们提升了一种特质而另外的

特质也得到提升的时候，就会知道我们为提升代码整体质量作出的努力处于正确的道路上。

我个人喜欢专注于内聚性。对我来说，内聚性是容易察觉和修改的，专注于内聚性帮助我根据事物的行为进行命名，让我的代码更容易阅读。我发现当我让代码更内聚的时候，代码的其他特质也提高了。你也可能更喜欢关注封装性或自主性。

无论你关注哪一个，如果你提升一个特质时发现其他的特质也得到了提升，你就知道自己的方向是正确的。

9.7　今天的代码质量提高会为将来带来速度的提升

我时常告诉各个团队，提高日后开发速度的方式就是现在提高代码质量。关注代码质量会保证我们编写的软件清晰且容易阅读。

我们大脑的特点让我们可以理解具备这些特质的信息，所以具备这些特质的代码更容易阅读和维护。高质量的代码帮助我们在修改软件的时候不会弄得一团糟，而且让项目具备伸缩性。

高质量的软件同样也更容易调试。这有助于快速交付且让代码容易维护，降低代码所有者的整体成本，而且收效迅速。

Ward Cunningham[①] 提出了“技术债”一词，用来表达如果开发者在代码中欠缺考虑最后会发生什么。没有什么比“技术债”更能拖慢开发进程影响预估的了。本来一个小时就能完成的事情，需要花费一天甚至更多时间来完成，有时候添加一个功能需要修改大量代码。增进代码质量可以降低修改带来的成本，并且让预估更加准确。

有时开发者必须追求速度，有时速度意味着粗心大意。如果你必须抄捷径，那么应该事后进行清理，避免以后造成更多的麻烦。当你采用那些有助于代码质量的实践的时候，花不了多少时间就可以编写出高质量的代码，会在以后很长时间内节省时间。当然，你必须多输入一些花括弧来在不同的类之间进行封装，但是打字速度什么时候是软件开发的瓶颈呢？

我有位朋友是专业厨师，每晚负责完成两百多道菜。“我没有时间去弄得一团糟”，因为他知道快速的工作就是整洁的工作。他的工作台永远干净整齐。他的袖口永远干净。他工作得越快，工作越整洁。我们也应该如此。

你也许在一本老书或者上过的计算机课中听过这些代码特质，如“内聚”“耦合”“封装”这样的词语。我发现这些想法是许多优秀的软件的核心。那些最成功的组织都非常迅捷，可以一直响应市场变化，给客户提供最优质的软件。他们的成功是建立在高质量软件基础上的。他们构建了重视代码质量的文化，其他的优势就随之应运而生了。我认为这是他们成功的关键要素之一。

① http://c2.com/cgi/wiki?WardExplainsDebtMetaphor

一旦你知道了如何编写CLEAN的代码，保证你编写出高质量、可测试的代码的最好方式是先编写测试，正如下一章中我们将要讨论的。

9.8 让我们付诸实践

以下是把这些想法付诸实践的方式。

9.8.1 提高代码质量的 7 个策略

人们对代码质量的定义各有不同。有人把质量定义为满足客户需求，有人把质量定义为执行速度快，还有人把质量定义为没有错误。这些都是好事情，但这些都是表象，我们应该如何着手编写高质量代码呢？以下策略有助于提高代码质量。

明确代码质量的定义

软件的质量和实物商品的质量不一样。正如烘焙师不能胡乱混合原料扔进烤箱然后期待做出一个蛋糕，理解组成优质代码的“原料”十分重要。优质的代码有什么特点？优质的代码应该清晰、容易理解、容易扩展。

对基本的实践达成一致

我们不仅要对“质量”的定义达成一致，还要对追求质量的实践达成一致。采用敏捷过程会有所帮助。比如，关注功能的构建，保持开发中心聚焦于给每个方法内聚的目标。

放弃完美主义

伏尔泰说过“不要让完美成为优秀的敌人”。我们中大部分人都知道在软件中是无法实现完美的，不会有这样的妄想。但是当我们不知道自己的代码会被如何使用的时候，就会认为只做到优秀是远远不够的。确立明确的验收标准可以帮助我们构建刚好所需的功能，而无需过度“镀金”。

理解取舍

开发软件需要在一定条件下做出一系列的取舍。如果理解这些取舍意味着什么，可以帮助我们针对当前情况作出更好的判断。失之东隅，收之桑榆，理解其中的取舍可以帮助我们构建总体上更优质的产品。

用“什么”来隐藏“怎么”

封装具体的实现细节，提供一个接口给调用者，描述他们要**什么**，调用者不必关心如何实现。这让我们可以自由修改实现的细节而不必担心影响调用者。

9

良好的命名

第一份文档（也是最重要的文档）就是软件本身。根据实体和行为“做什么”命名，而不是根据“如何做”命名。始终保持有意义的名称和隐喻。这让软件更容易理解和维护。避免使用缩写词和首字母缩略词，代之以“驼峰命名法”拼写。不用担心它的长度，集成开发环境（Integrated Development Environment，IDE）都有自动补齐功能，你只需要输入几个字母然后从列表中选出所需要的就行了。让名称具有描述性，使用主动语态，正面陈述，反应其调用之后对系统的影响。

保持代码的可测试性

未经测试的软件包含着未知的风险，所以测试是十分重要的。但是除了验证代码是否正常工作之外，更需要保证代码的可测试性，因为可测试性高的代码也是高质量的代码。

高质量的软件不是自然产生的，也不是通过“六西格玛”或“瀑布模型”这样的重量级流程创建的。软件质量来自于对问题的仔细思考以及建模的准确性。关注软件质量可以帮助我们通过集中决策来摆脱冗余，让方法和类更专注，让行为处在正确的地方。结果就是代码库变得容易维护和容易扩展。

9.8.2　编写可维护代码的 7 个策略

可维护代码是那些容易理解且容易修改的代码。它设计清晰、实体命名合理，开发者修改的时候不会心生恐惧。可维护代码不是偶然产生的。虽然需要付出额外的注意力，但是对维护代码的人来说却意义重大。以下是编写可维护代码的7个策略。

确立代码的集体所有权

代码的集体所有权意味着团队中每个人都可以维护任意部分的代码，即使并不是他们编写的。团队应该确立共同的代码规范，让人没办法通过阅读代码来看出是谁编写的。这会保持代码风格一致且容易维护。除了统一代码格式之外，团队也需要统一领域模型和开发实践，使用同样的词汇来描述设计。

积极重构

重构是编写代码的关键步骤，而且贯穿整个开发过程。开发者应该在编写代码和新功能完成时进行重构。重构不是编写糟糕代码的借口。重构让代码更健壮，便于以后维护。

坚持结对编程

在团队中促进知识传播的最好方式是结对编程。每天和不同的人结对，直到找到和你最合适的，但继续偶尔和其他人结对，以保持不间断的互相学习。有的团队对所有的任务都进行结对，你应该至少在设计、编码、重构、调试和测试的时候结对。

频繁的代码审查

即使有了结对编程，代码审查也是十分有价值的，因为它给了没有和你结对的人一个机会，去阅读你的代码并给予反馈。代码审查应该着重于决策背后的原因，以及讨论设计选择和取舍。

学习其他开发者的风格

阅读其他人的代码，学习其他开发者是如何编码的，这是一个提高开发者技能的好方式。几乎每个开发者都有自己的风格。学习其他开发者解决问题的方式能让你自己更优秀。

不断学习软件开发

二十年前只有少数几本软件开发者必读的书。时至今日已有了几百本。专业的开发者需要持续学习。医生每周花8~10小时在他们领域相关的阅读上。软件开发者也应如此。

读代码，写代码，练习编码

史蒂芬·金在《关于写作》中说，想要成为优秀的作家需要大量的阅读和大量的写作。当人们问Henny Youngman如何能做到在卡内基音乐厅演奏的时候，他的答案是："练习，练习，再练习。"对于软件开发也是如此。阅读他人的代码，编写代码，不断练习。

整洁的代码更容易维护。平均来说一次编写的代码会被阅读十次，所以尽力保持代码整洁是有意义的。当这些实践成为习惯之后，你会发现基本不需要花什么力气，更具维护性的代码却很快会得到收益。

9.9　总结

编写CLEAN的代码，或者说内聚、松散耦合、封装良好、自主、没有冗余的代码。CLEAN的代码是高质量的代码。

本章中心思想如下。

- 高内聚的代码减少副作用。
- 松散耦合的代码更容易测试。
- 封装良好的代码容易扩展。
- 自主的代码让软件更模块化。
- 没有冗余的代码降低维护成本。

如果把优秀代码定义为容易理解、容易维护的代码，我们就能找到相应的代码特质来帮助编写优秀代码。这些代码特质看上去都是小事，但它们会累积起来，关注这些细节会让我们构建出更容易维护的软件。

第 10 章

实践6：测试先行

测试驱动开发还没死？

我们接下来要讨论的实践是测试驱动开发（Test-Driven Development，TDD），有些人声称TDD已经死亡了。他们说，这个想法很好，但在实际中并不管用。他们用“测试引入的损伤”来形容那些过度的测试或依赖实现的测试，这些测试最终成了负担而不是资源。

他们说的没错。

我见识过“测试引入的损伤”。开发者若不知道何时该停止编写测试，就会出现这种情况。

测试先行的一个主要好处就是，在我们必须修改现有代码的时候，这些测试给了我们质量保证。但是，如果开发者编写了过多的测试，或者编写了依赖实现的测试，这些测试修改起来比修改代码还困难，不仅没有让代码容易修改反倒成了累赘，让修改代码变得更困难而且更耗时间。

有的TDD实践者告诉我，编写测试直到你觉得无聊为止。他们的理由是当你达到无聊的阶段，往往就不需要更多的测试了。这种方式的问题是，我们每个人对无聊的界定都不同。“直到无聊为止”没有明确告诉我们需要编写多少测试。

测试是标准，测试定义行为。

从这个角度看待测试，可以让你明确知道针对某一行为需要编写多少测试，以及如何通过实现这些测试来创建行为。通过测试来驱动开发，不仅可以让你知道为了让代码正常工作需要多少测试，需要什么样的测试，还可以保证实现变得集中，让你知道什么时候算开发完成，可以进入下一个任务。

正如其他复杂的活动一样，只有少数几种正确的做法，却有成千上万种会搞砸的方式。使用方式不正确造成的后果不能归咎于TDD本身。

我提倡这样的TDD方式：它可以明确指导我们正确使用TDD，避免测试引入的损伤。只针对需要构建的行为编写测试，然后编写不多不少刚好能让测试通过的代码。

10.1 测试的种类

在软件开发中我们将许多不同的事情称为“测试”。它们各不相同并且有不同的目的。在深入探讨TDD之前，我们先讨论一下在软件开发过程中用到的不同种类的测试。

10.1.1 验收测试 = 客户测试

客户测试，或者说**验收测试**，明确了用户故事中的行为，帮助软件开发者和产品负责人（或者客户代表）进行真正的沟通。这种沟通可以是正式的，开发者编写出可以让自动化框架执行的测试。也可以是非正式的，只是简单地在故事卡片上写出用户故事的几个例子。

验收测试帮助开发者理解那些边界用例在哪里：针对一个场景都有哪些异常，好让他们知道如何处理额外的执行路径。通过对验收标准进行定义，验收测试回答了这个问题：“我怎么知道什么时候算做完了？”

顺便提一下，这是开发者需要回答的最重要的问题之一。你也许不清楚自己编写的代码会在什么样的场景下执行，因此会倾向于过度开发。在有限的时间内，如果你在某些部分过度开发了，意味着在另外一些部分就会开发不足。知道什么情况算“完成”让你可以在需要的时候继续前行，让你有满足感和完成感，这些感觉都是开发者梦寐以求却很少能体会到的。相反开发者总会不解：“我是不是遗漏了哪个用例？我的程序够健壮吗？”

如果有了验收测试，执行的时候一路绿灯，你就可以轻松地继续后面的工作了。如果还需要更多的功能，就等那些功能明确了之后在下一个迭代中开发。在软件开发中有很多歧义，对于软件开发者以外的人来说难以应对，软件开发者喜欢非此即彼的事情：“我有没有覆盖某一用例？”

有许多进行自动化验收测试的工具。可以通过Gherkin这样的语言来用类似英语的方式编写测试，Gherkin基于三个准则——环境、条件、结果。

> 在特定的场景下，如果触发了一些事件，我们会看到这样的结果。

“特定的场景”是你为测试做的前期准备，是搭建好测试需要的先决条件。“触发的事件”是开关，在此引入需要测试的行为。“结果”是你对实际执行结果和期望结果进行比较，然后就可以知道测试是否通过。

记住，我们需要“二选一”的结果：通过或未通过，而且测试应该是自动的。

关于验收测试驱动开发（Acceptance Test-Driven Development，ATDD），我最喜欢的两本书是Ken Pugh的《精益敏捷验收测试驱动开发》[Pug11] 和Gojko Adžić的《实例化需求》[Ad 11]。

10.1.2 单元测试 = 开发者测试

单元测试并不测试整个用户故事，而是测试相对来说更小的单元。TDD中开发者用单元测试

来驱动开发。

单元测试也可以当作内部文档来使用，这能节省很多时间，并且提供了整套的回归测试，在以后修改代码的时候可以捕捉异常。这些测试现在可以自动化执行，当你执行一个测试套件的时候，你编写的所有测试都会被执行。

这一章重点关注单元测试，但我希望你同样也了解软件开发过程中其他种类的测试。

10.1.3　其他测试 = 质量保证测试

除了验收测试和单元测试之外，软件开发过程中还有其他种类的测试。其中大部分都是质量保证过程的一部分，比如**集成测试**，用来测试工作流中不同组件的交互。和单元测试不同，集成测试不会把所有依赖用模拟对象替代，而是使用真实的依赖测试组件之间的交互，所以集成测试更容易出错也更慢。复杂的工作流需要许多集成测试，这会严重拖慢构建的速度。

我倾向于尽可能多地使用单元测试，和依赖保持隔离。这样会让测试变得更快、更简单。

我知道有些团队使用工具来模拟用户输入以测试他们的功能。这些工具在测试世界里有它们的一席之地，在有的场景下非常有用。但是，如果这是你测试系统功能的唯一方式，那么也许有些设计上的缺陷需要重新思考下。有时候你被现有的系统束缚住手脚，必须要做出取舍。但是，只要有可能，你就应该尽量去编写那些可以被单元测试验证的代码。

有许多测试工具和方法可以保证代码质量。为了方便叙述，我们把测试分为两类：

- 作为待发布版本需要进行的测试
- 所有其他的测试

尽可能地将测试自动化是非常重要的，将待发布版本的测试自动化尤为重要。正如优秀的代码会有最少的依赖，让出错的可能最小化，我们的软件开发流程也应该有尽可能少的依赖。如果构建成功与否依赖于人工干预，则引入了一个依赖：人工的不确定性。

在传统的质量保证流程中，对于待发布版本的验证通常有许多人工步骤。这提高了发布的成本，迫使我们以更大的批次进行构建。

然而，通过自动化测试，你基本上可以移除构建过程中所有的人工干预，进而保持系统一直处在可发布状态，可以不断向其中添加特性。有些软件无可避免地需要人工干预。比如说我们没办法在足不出户的情况下完整测试GPS软件。但是你可以将这些组件隔离出来进行模拟，以便这类行为不会阻碍你的关键路径。

10.2　质量保证

质量保证（Quality Assurance，QA）测试有许多种形式。

- “组件测试”体现各个组成单元之间的配合情况。
- “功能测试”体现所有组成单元在一起完成整个端到端的行为。
- “场景测试”体现用户和系统的交互行为。
- “性能测试”验证这些情形：“这个系统能承受很大的负载吗？我们进行过独立测试，但是如果百万级用户进行并发请求会怎么样？”
- “安全测试”验证代码的脆弱程度。

等等。

项目所需的质量保证测试因其面临的挑战不同而不同。测试心脏起搏器的方式和测试社交应用的方式完全不同。

质量工程（Quality Engineering，QE）在软件开发中扮演着一个特殊的角色。微软、亚马逊和谷歌这样的公司意识到，他们需要以工程学的形式关注产品质量，所以他们辞退了很多只会手动执行脚本的人，代之以会执行自动化测试的开发者，而且这样做有所回报。

这和到目前为止讨论的其他问题是一致的，我们能够将一个用户故事从头到尾尽快完成，因为“完成”包括验证其正常运行的自动化的测试，验证的过程应该尽可能快速。

10.2.1　测试驱动开发不能取代质量保证

软件产业推崇对代码进行测试，但是大部分人还是习惯后编写测试，最后往往会花费更多精力。

整个团队都在最后进行测试或质量保证，就如同在出现储蓄信贷丑闻**之后**才送交审计，或者飞机失事**之后**打开黑匣子。我知道其中的意义，至少可以收集信息避免以后发生坠机，或者进行责任认定。但是审计组并不能把人们的钱追回来，调查组也无法让飞机坠毁的遇难者死而复生。

将质量保证推迟到开发周期中临近发布的最后阶段只会给你带来坏消息。如果你找到了一个bug，已经太晚了，从bug产生到它被发现并修复的时间越长，成本越大。

测试驱动开发有助于质量保证进行自动化的回归测试，但同时也需要回过头添加测试去覆盖一些额外的测试用例。

如果你开始将测试驱动开发视为对行为的**定义**而不是对行为的**验证**，你就会对所需的测试有更清楚的认识。这样进行测试驱动开发，是为了给你安全地清理代码提供简洁又有意义的基础测试，但它不能取代质量保证工作。

10.2.2　单元测试不是万能的

有些事情适合单元测试，而有些事情无法在单元的层次体现，但又需要进行测试。比如，端

到端测试可以让你知道应用程序是否能如预期一样从头到尾正常运行。

多数系统都含有并非通过测试驱动开发的代码，这样测试起来就会有难度。这些代码包括：和用户界面交互的代码、和数据库交互的代码、多线程代码，以及很多其他的代码。

尽管如此，我还没有见过即使下决心修改设计也依然无法测试的功能。

但是，有时你无法修改设计，比如使用已经编写好的代码，或者使用特定的现成软件包。如果你可以控制执行环境，通常架构层次的调整可以让原本难以测试的代码变得容易测试。而这样做之后，你会看到系统成本变得更低，更容易维护，更容易扩展；都是好消息。

无论如何，单元测试是一个好的开端。我们在后面会看到，当进行测试先行开发（即先写测试，然后编写代码，保证测试通过）的时候，开发者得到的不仅仅是一套自动化的单元测试，还能构建出具有灵活性的系统。

10.3　编写优质测试

我见过的大部分开发者都自认为知道什么样的测试是优质的，并且知道如何编写。但是在我的经历中，只有少数人真正理解如何编写优秀的单元测试。和多数事情一样，编写单元测试是需要学习的技巧。

许多管理者（以及软件开发者）对于TDD是什么有着奇怪的理解。很多困惑源于他们讨论TDD的方式。实践测试驱动开发的方式有很多，我推荐"测试先行开发"：开发者先针对一个功能编写测试，然后实现那个功能让测试通过。从测试角度（由外而内）到代码角度（由内而外）反复切换，可以给我们所需的反馈，让开发进程变得更稳定。

我知道有的人先编写出所有的测试然后再编写应用程序代码。我并不认为那是真正的测试驱动开发。那样做是有些好处，但是并不如测试先行开发中获得频繁的反馈收效大。

在所谓的"测试后行开发"中，先编写代码再编写测试。这样做有明显的好处：可以得到一套自动化的回归测试，可以在对系统进行修改的时候执行以保证系统的其他部分依旧正常工作。但是，在编写代码之后编写测试往往会发现代码难以测试，需要大量的清理来让其可以测试，这可能会成为一个大工程。最好是一开始就编写出可测试的代码，而编写可测试代码的最好方式是先编写测试。

先编写测试的另一个显著益处是，你只需要编写测试覆盖的代码，所以测试覆盖率永远是百分之百。如果采用我推荐的测试先行开发，则没有一行代码不是为了让测试通过而编写的，也就是说所有的代码都被测试覆盖。

通过编写代码来让测试通过，这能保证你写出的代码具有可测试性，因为针对已经有的测试写出难以测试的代码反倒很困难。作为软件开发者来说最大的挑战之一是，我们很容易编写出难

以测试的代码。所以，当事后进行测试的时候，才发现必须重新设计和编写很多地方。进行测试先行开发可以防止开发者陷入这样的境地。

10.3.1 这不是测试

在我编写实现代码之前编写单元测试的时候，我在测试什么？还什么都没有，所以我编写的不可能是测试，因为明显还没有可测试的东西。

如果不是一个测试，那是什么？把它当作一个**假设**来看待。

在编写先行测试的时候，我所做的是我对于需求理解的**假设**。我对如何调用一个服务以及我希望什么样的结果进行假设。我对完成要求进行假设。

在我编写完代码让测试通过之后，因为已经有了一些行为，这就变成了真正的测试：可以通过测试来验证这些行为了。这些测试在整个软件的生命周期中持续产生价值，验证我们所做的任何变更都没有影响到代码，它们依然如预期的那样工作。

我们所谓的“测试”其实扮演了两个角色。一方面它们是假设，或者说是对代码行为的规范。另一方面，它们也是那些创建之后有持续价值的回归测试，可以为我们验证代码是否如预期的那样工作。

从某种意义上说，测试是**传感器**，就像你车上的引擎灯一样。它们一直在那，如果新添加的代码让它由绿变红，就是告诉你有些地方出错了。

10.3.2 以行为作为单元

TDD对于我们解决问题的思考方式有很深的影响，但是TDD可能被误用。我们用的许多词语都具有误导性，所以很多开发者——甚至是那些资深开发者——都可能会迷惑。说到“单元测试”，许多开发者认为“单元”是一段代码。

但并不是。

我们说的测试并不是狭义的测试，单元也不是指代码的单元。当我们说“单元测试”中的“单元”的时候，并不是指很多开发者理解的那些方法、类、模块、函数等的实体。

单元是指一个**行为的单元**：一个独立的、可验证的行为。它必须对系统产生**可观察的影响**，而不和系统的其他行为耦合。

理解这点十分关键。

“单元”这一词语用来强调一个行为不依赖系统中其他行为单元。这并不是说每个类都需要有一个测试类，或者每个方法都要有一个测试方法。单元测试意味着每个可观察到的行为都应该有一个相对应的测试。

代码始终应该产生可以观察到的行为。

现在对设计进行整理，如果行为不变，则不需要添加新的测试。即使需要添加新的类或方法，如果行为一样，测试也不应该受到影响。此时不应该引入新的测试。

这也许听上去简单，但是却让很多开发者在开始实行测试驱动的时候迷失。我听说有的开发者担心，每次建立一个新类或新方法的时候，他们都需要编写新的测试，然后代码越来越庞大，测试也越来越多，最后变得难以想象。

那是因为他们做的方式不对。

一个“单元”代表的是一个**行为**，如果**行为**不变，**测试**也不应该变化。

先行编写测试的目的是帮助开发者高效编码，保证以后能根据需要**轻松**清理代码。你不希望在清理代码的时候让测试成为你的负担，那会大大降低自动化回归测试带来的价值。要编写可以对行为进行定义的最小数量的测试。

10.4　TDD 可以提供迅速的反馈

开发软件成本最低的方式就是避免bug发生，除此之外就是尽快发现bug，**立即**交给最初编写代码的人或团队修正，而不是在临近交付的时候由另外的团队处理。

俄罗斯生理学家、心理学行为主义学派创始人Ivan Pavlov告诉我们，如果想要“刺激–响应”过程形成持久的精神影响，响应必须紧随着刺激。“敲击这根杆，然后迟早我们会喂你”对狗来说没有效果，同样的道理适用于软件开发者，如果拿**自己**和狗对比的话。

如果我犯了一个错误三个月后才发现，我没法把前因后果对接上，今后很可能会犯同样的错。到那时候，我已经深处于另一个项目，必须停下手中的工作回过头去处理一个根本想不起来的bug 。

这就是为什么事情得不到改善，因为开发者没有得到迅速的“刺激–响应”。测试驱动开发可以提供这样的迅速反馈。

10.5　TDD 可以为重构提供支持

从计算机发展的早期开始就一直或多或少地实行重构，在Martin Fowler出版《重构：改善既有代码的设计》[FBBO99]之后，重构才真正引入软件开发理论体系中，在书中他将重构定义为“在不更改外在行为的前提下，对代码做出修改，以改进程序的内部结构”。每个开发者都应该阅读这本书，理解这本书，其中包括了许多优秀的编程实践。

重构给了开发者机会，让他们的代码更容易使用和维护。有时，当开发者沉浸于一个功能的实现之中时，他们倾向于马马虎虎的命名。有时，开发者在实现一个行为的时候，不确定该如何

命名某些事物。为此，回过头审查代码非常有必要，可以找机会优化命名，让代码更容易维护。

在敏捷软件开发中，并非在一开始就得到全部的需求，而是一边设计一遍理清需求，这很容易做出一些错误的选择。即便如此，迭代构建也比事前确定所有需求要高效得多。如果你做出的错误选择限制了你根据当前需求对设计做出相应的扩展，可以重构代码而不会降低已有设计的质量。

但是重构也可能是危险的，因为多数的代码互相关联，一处的修改会引发其他地方的bug。

有测试支持的代码重构起来会更安全。如果你犯了一个错误，很可能会让你的测试之一失败，所以你会立刻发现这个错误并马上修改。

10.6 编写可测试的代码

我曾经对某个大客户——某互联网巨头——做过TDD的培训课程，在周一课程开始前，大约是早上8:45左右，一位资深经理走进来说："我听说你在这儿教TDD课程。"

我说是的，他回答："我们这里不用TDD。我们不想用TDD。你为什么教我的人TDD？这么做不对。"

我对自己说，**冷静**。**深呼吸**。然后说："实际上，我不关心你的开发者是不是进行测试先行开发。我关心的是他们编写出可测试的代码，而TDD是最有效的方式。"他对我的回答很满意。他可以理解那样做的意义。

可测试性和代码质量之间有紧密联系。TDD不会替你做出设计，但是可以提供基本框架，用来支持可以产生优质设计的自然而然的思想方式。将我们思考的多种方式转变为测试驱动开发周期中的若干个阶段，可以让问题更容易解决。

我岁数不小了。身为一个软件开发者已有三十余年。在年轻的时候我有着超强的记忆力。我可以记住各种细节——难以置信的细节。我可以将我脑海中的代码编写出来。我可以在脑海中构想出数千行伪代码然后编写出来。

但是随着年龄的增长，我的脑力开始衰退。好消息是，增长的智慧可以取代集中度、注意力、耐力和专注程度。TDD支持我们自然的思考方式，帮助我们集中注意力，产生更好的结果。它让过程变得更简洁，从而更容易解决问题。

即便如此，要实现TDD还需要许多技巧，你可能会发现自己面临下面两种情景之一：

"哦，我可以针对它来编写测试——我理解它。"

或者"根本不可能——我做不到。"

为什么会这样？

当我陷入第二种情景之中，没法进行测试，则说明我有些设计问题需要重新进行思考。

理解了这一点，这些反模式，这些随之而来的坏事，也就没那么糟糕了。它们是关于如何重回正轨的线索。

我曾做过一些视频剪辑，每次有一个错误剪辑的时候，总是需要五秒之后才发现，而当时我已经进入另一个场景了，必须倒回出错的地方。但是，我不单单解决问题之后继续工作，还希望以后不再出错。我倾听那个错误的声音。

在TDD过程中我总有办法保持效率。我可以清理代码，或者为新的行为编写另外的测试：我可以把复杂的问题拆解为一些小问题。实践TDD就像有一个难度开关，当我卡住的时候总能将其调到“极度简单”，在其中积累足够多的自信后再调高难度。而这一切都在我的掌控之中。

为了要写一篇博客或者教一堂课，我必须先休息好并准备一杯浓咖啡。但是我可以在半梦半醒的时候写测试先行的代码。也可以一边和我妻子看电影一边写测试先行的代码。这就是这个流程的威力所在，对于我们中那些依赖脑力来构建复杂代码的人来说，是一个莫大的解脱。但这不仅仅是因为我的懒惰，我真的可以产生更好的结果，这就是为什么我会如此推崇它。

10.7　TDD 也会失败

尽管TDD非常有价值，但我也见过它在一些公司里彻底失败。有位客户曾经确信地跟我说TDD不管用。他说他们不得不抛弃TDD。当我问他为什么的时候，他说他需要花一天时间来清理代码和一个礼拜来清理测试。

他们时间紧迫，所以必须做出选择：是坚持TDD而导致项目失败（然后看着整个公司破产），还是放弃TDD。如果有人面对这样的抉择，放弃TDD肯定是正确的选择。

不要在发布前才开始实施TDD。不要在开发者承受不起的时候给他们添加新的学习曲线负担。

这位客户确实有地方做错了。他们引入了代码质量管理，引入了CLEAN原则，引入了优秀的开发原则，等等，但没有将测试也当作代码，所以他们的测试中有大量冗余。他们把测试当作附属物而非系统中的一部分。

他们拉起了质量保证的大旗，然后说“测试越多越好”。于是他们编写了过多的测试，而且测试是针对实现（他们的做事方式），而不是针对接口（他们希望的结果）。结果就是，在他们清理代码的时候发现清理测试变得非常困难。记住，单元测试应该能辅助你对代码进行清理，所以编写测试的时候一定要考虑可支持性。

这位客户还跟我说：“每次执行测试都需要好几个小时，所以我们不经常执行测试。它们怎么会这么慢？”

他接着说："嗯，我们得先连接到数据库，然后需要用数据库来……"

然后我说："你不是Oracle的。你只是使用Oracle数据库，但是不为Oracle工作。他们在街对面。你为什么要去测试Oracle的代码？"

"我们的代码和数据库有交互，所以需要连接到数据库才能进行交互啊。"

这可不是编写单元测试的正确方式。

单元测试仅仅用来测试**你的**代码单元的行为。

如果代码和其他系统有交互，你需要将它们进行模拟而只测试你的代码。我向那位客户展示了多种使用模拟来编写可测试代码的技巧，如分流、依赖注入、内切测试以及其他技术。

这些技术都可以帮助你只对你想测试的代码单元进行测试。如果你只测试需要测试的那一部分，测试速度就会变得很快。

10.8　如何将 TDD 引入团队

有人问我TDD能否由单独的开发者（或者开发团队）自主引入他们的公司，而不需要进行某种正式交割，或者需要管理层介入这种单纯的技术范式切换。

事实上，许多实施TDD的公司就是这么开始TDD的。有时管理层会说："同志们，只要你们觉得是可以帮助你们构建高质量软件并且遵守交付期限的事情就可以放手去做。"而有些公司则会说："什么？你需要两倍的时间——编写两倍的代码？你疯了吧！"

所以，TDD的实施非常依赖管理层对TDD的看法如何。我们同样也会面对开发者的成见。这些成见各有不同。管理者和开发者都有自己不愿意执行TDD的原因，但是，当每个人都见识过执行TDD的效益时，他们都会开始跃跃欲试。

学习如何正确进行TDD有不止一种方式。它不是简单的复制粘贴，而是源于一些共识，如果想从实践中获得收益，就必须遵循这些核心规则。

10.9　成为测试感染者

有的开发者告诉我，他们对于先编写测试感到不适应。对许多刚开始TDD的开发者来说这是最大的挑战，因为我们之前的经验一直都是关注于直接编写实现。只要把一副键盘放到开发者手中，他们就开始编码了。我花了相当长的时间才克服这个习惯开始先编写测试。但是开发者是**做什么**和**怎么做**之间的桥梁。

开发者应该从**做什么**开始，因为做什么就是所谓的接口，也是测试点所在。测试应该是关于**做什么**的。从**做什么**开始入手通常都是好主意，因为这样可以帮助我们防止实现相关的信息泄露

给系统的其他部分，让代码更加专注而且封装性好。

首先，要对**做什么**进行详细定义，说明你调用的方法会是什么样的：方法名、输入参数、返回值。然后再关注**怎么做**：这个方法如何执行工作。所以，实际上就是简单后退一步思考一下：**在真正开始编码之前，我要创建什么**？然后理解它的上下文。多数情况下，这是我们解决问题时应该采用的方式。

当你打破直接编写生产代码的习惯（仅仅是个习惯而已）的时候，你就自然而然会以**测试先行**的方式思考了。测试驱动开发的先驱之一Erich Gamma[①] 发明了一个词：**测试感染者**。

当你发现了TDD的价值，非得使用TDD不可的时候，你就成为了“测试感染者”。我就是测试感染者。你给我多少钱也没法让我以非测试先行的方式编写代码。

10.10　让我们付诸实践

以下是把这些想法付诸实践的方式。

10.10.1　进行优质验收测试的 7 个策略

验收测试可以从任何层次进行，从用户故事到任务。它们可以用Cucumber或SpecFlow这样的测试框架来进行标准化，或者可以简单记录在故事卡片旁边。验收测试告诉团队什么情况下算完成了一个功能，帮助每个人对要构建的东西的理解达成一致。以下是进行优质验收测试的7个策略。

明确构建目标所产出的价值

编写验收测试迫使你明确理解构建的目标和它在系统中的影响。仅仅是让产品负责人和开发者直接进行对话就已经很有价值了，而验收测试还可以让人们对如何优化他们的产物进行思考。

理解为谁而做以及他们为什么需要

不仅仅要知道用户需要**什么**，开发者还要知道为**谁**而做以及他们**为什么**需要。这可以帮助开发者找到完成任务的更好的方式，从而让产品更容易维护。将用户人格化，给他们背景故事。明确功能是为谁而做，以及该功能的目的，这样可以帮助开发者让那个功能产生更多价值。

将验收测试自动化

将验收测试自动化对于开发者和客户来说都是非常有价值的。通过实例引发讨论，对验收标准进行定义，可以让每个人都对将要构建的目标达成共识。

① Gamma, Erich.“Test Infected.”http://junit.sourceforge.net/doc/testinfected/testing.htm

定义边界用例、异常、次要路径

验收测试也可以定义代码中的次要路径。提前定义这些边界用例，可以帮助开发者优先处理最主要问题，帮助开发者关注哪些地方可能会出错，从而构建更健壮的程序以应对这些问题。

用实例来充实细节和展示不一致

通过使用一个功能的实例展开工作，对于理解功能相关的实现问题是个很好的开始。实例帮助你对事情进行具体地思考和讨论。通过具体实例开始工作。一旦做过一定数量的具体实例，你就可以开始对其进行概括，用抽象的代码进行处理了。

用验收标准来拆分行为

每个验收测试都有一个独立的验收标准，通过或者不通过。不同的行为应该有不同的验收标准。这样可以促使构建出的功能关注一个独立的验收标准，而不依赖其他功能。

保持每个测试的唯一性

每个验收测试都应该唯一，而且和其他验收测试独立。唯一的验收测试关注于单一的验收标准，确保代码没有冗余。

验收测试告诉开发者他们需要构建什么，更重要的是，知道什么时候算完成。知道一个功能的完成标准，可以防止开发者过度开发，并且给了他们继续进行下一个任务的信心，无需怀疑自己。

10.10.2　进行优秀单元测试的 7 个策略

单元测试要么成为资源，要么成为累赘，取决于重构时单元测试产生的价值。如果你的测试依赖于实现，则在重构代码的时候必须要跟着一起重构，这样就会花费更多的时间而不是节省时间。以下是进行优秀单元测试的7个策略。

从调用者的角度出发

永远从调用者的角度开始对一个服务进行设计。想着调用者需要什么，必须传入哪些参数。先杜撰一个方法签名，在设计的时候再进行重构。

用测试定义行为

用编写测试来高效驱动功能开发，这样可以帮助你进行设计，会给你一组优质的回顾测试。这些测试会成为系统中的**活文档**。点一下按钮就能确认它们是否是最新的。

仅仅编写能体现区别的测试

让测试**唯一**。每个单元测试都应该推动开发进度前进，在系统中创建一些新的可观测的行为。如果你这样做，你的测试就会是唯一的。

仅仅编写可以让测试通过的代码

如果你需要编写代码，先编写一个失败的测试，然后编写代码让测试通过。这个规则保证你的代码被测试覆盖。

用测试来构建行为

有许多通过测试构建行为的方法。你可以从**快乐路径**开始然后处理异常，或者从错误用例开始再到快乐路径。选择的方式根据情况不同而不同。

对代码进行重构

随着需求的展开和理解的深入，重构代码保持可维护性十分重要。保持代码简洁且健壮对于迭代式开发十分关键。

对测试进行重构

如果你针对行为而非实现进行测试，当你重构的时候就不需要对测试进行添加或修改。测试会为你的重构提供保障。但是测试也是代码，同样会被糟糕的代码质量所累，除非你花时间优化测试代码，让它更健壮、更容易使用。

一组优秀的单元测试可以防止系统衰退，为开发者重构代码提供安全保障。使用单元测试来定义行为，可以清楚展示代码的使用场景，成为权威的**内部**文档。用测试先行的方式进行开发，可以让软件的质量更高且维护成本更低。

10.11　总结

先编写测试，然后仅仅编写可以让测试通过的代码。这让你构建的软件更专注且更容易测试。测试应该对你的重构提供安全保障，测试代码也需要和产品代码一样保持CLEAN。

本章中心思想如下。

- 如何进行以及**为什么**进行测试先行开发。
- 如果编写过多的测试或编写实现依赖的测试，那么测试先行开发就出了问题，将阻碍你重构代码。
- 测试应该支持重构，为此仅编写对要构建的行为进行定义的测试。
- 进行测试先行开发有助于保证代码质量，但不能取代质量保证。
- 测试先行开发需要先编写失败的测试，然后编写刚好足够的代码来让测试通过。之后根据需要进行重构，接着再编写另一个失败的测试。

如果使用得当，测试先行开发可以帮助开发者编写容易测试和容易维护的代码，但是，如果使用不当，TDD会成为累赘而非资源。

第 11 章

实践7：用测试描述行为

测试可以给你及时的反馈，让你知道所做的事情对系统的其他部分是否产生了不利影响，如果真发生的话，你就可以自动回滚那些修改，一切都是那么容易。通过单元测试来对代码进行独立验证，可以即刻发现各种逻辑错误。很多难缠的bug让我们通宵达旦、两眼通红地盯着调试器，现在这些bug可以通过单元测试在编写时发现，而无需在后期进行排查。

开发者可以立刻知道哪些代码能正常工作哪些不行，这改变了软件开发的方式。有了即时的反馈，你可以很快理清头绪，可以进行激进的实验，因为你知道测试可以帮助你捕获错误。一旦建立了即时反馈机制，团队中的开发者就可以把它当作构建更优质软件的学习工具。诚然，学习现有的优秀编程实践会节省很多时间，但我有信心，开发者可以利用测试得到的即时反馈探索出更多的优秀编程实践。

我有一次买了一辆价格很优惠的二手车，但是有些电路上的问题。我找了三个不同的修理工，都没找到问题的根源。前灯总是熄灭，转向灯也不管用。他们更换了前灯，转向灯也能工作了，但坚持不了多久。最终，第四个修理工找到了问题，实际上问题不只一个，有**两个**问题。

转向灯的电路有两处短路，这就是之前三个修理工都没能发现的问题的真正根源。诸如此类的复合型问题，是由于多个问题夹杂在一起产生的，它们如果出现在软件里面，几乎不可能顺利地排查。一旦这种情况发生（一定会发生），就需要数小时甚至数天去跟踪解决。

根据Capers Jones在《软件系统成败模式》[Jon95]中所述，开发者通常要花费一半以上的时间修订之前所做的决定。这一切都消耗着宝贵的时间与资源，这就是测试驱动开发所要针对的问题。

所以当管理者或开发者跟我说，他们没时间在编写实现之前编写测试，我并不奇怪。因为没有实行测试驱动开发，所以他们没有时间实现测试驱动开发。但是我们所有人——开发者和管理者——早晚都需要从枯燥的活动中解脱出来。我们这个产业当中还没有全部问题的答案，TDD也不能解决所有问题，但它的方向确实是正确的。

TDD帮助我们对自己编写的代码有更好的理解，因为它能用一个具体的实例来给代码构建一个上下文，这正是我们最佳的思考方式。人类用具体的方式进行思考和构想，但编程语言常常用抽象来描述事物。我们必须进行这种转换。

测试是具体的需求，因为它们用特定的参数去执行代码。将需求具体化对于发现优秀实现方案来说十分有价值。如果你这样构建软件，就能从根源上消除大量潜在bug。

11.1　红条、绿条、重构

测试驱动开发有三个独立的阶段。我们称之为**红条、绿条、重构**，因为这些是你可以从单元测试框架中得到的明显提示。如果所有的测试都通过了，就会得到一个绿条。如果有些测试失败了，就会有个红条。

最开始编写测试的时候还没有可以测试的代码。无所谓成功失败——甚至都不能编译。如果使用现代的集成开发环境（Integrated Development Environment，IDE），你会得到类似这样的提示："我不知道你这段代码指的是什么。你希望一加一。我从未听说过一个叫 `add()` 的方法。那是什么？"

接下来需要做的是为产品代码做"存根"，好让测试通过编译。编写一个名叫 `add()` 的方法，但只是个存根。接受两个数作为输入参数，但仅返回一个 0，因为我们暂时不关心实现。

这是一个非常简单的例子，对两个数做加法用不着太多思考。但是，想象一下，如果它是"双倍余额摊销算法"或其他复杂的事情将如何呢？首先要确定如何命名。需要它做什么？希望得到什么样的结果？但现在暂时只返回一个虚拟数值。这就是所谓的"方法存根"。

一旦创建了方法存根，就可以对测试进行编译并执行了。当然，测试会失败，而且是通过红条来告诉你测试失败了。这时对红条的观察就是对测试的测试。测试是**允许**失败的，这点非常重要，因为测试不可失败比没有测试还要糟糕。

接下来开始编写能让测试通过所需的最简单的实现代码。完成之后就会看到绿条，说明测试通过了。如果需要的话，下一步是清理代码提升质量，以便代码日后更容易阅读和使用。同时也要对测试进行清理。然后继续第一步去开发另外一些东西，另一点意图或者行为。

在编写测试的时候其实是在说："你好代码，你能做到这些吗？"代码当然不能，所以测试失败。得到了否定回答，你说："好吧，让我来告诉你怎么做。"之后你编写完实现，然后说："你好代码，现在你能做到吗？"然后如此往复。

这样和你的代码进行对话非常美妙。一点一点，周而复始，红条、绿条、重构，周期循环。用这个方式构建整个系统。

对于外行来说可能有些单调，但事实并非如此。完全不单调。那是一首旋律，就像歌曲中的鼓点一样。

11.2　一个用测试先行来描述行为的实例

让我们详细观察一个实例。这个例子演示了如何针对特定行为创建合适数量的测试，进而驱动该行为的开发。

如果你不是程序员且看不懂源码，没关系。如果我的命名没问题的话，你应该可以理解代码的大概要领。即使不理解代码是如何工作的，你也应该可以体会开发者在构建软件时所要考虑的许多事情。

假设要建立一个具有姓名和年龄的 `Person` 类。用Java语言编码，使用Eclipse[①] 开发环境。

11.2.1　编写测试

首先编写**快乐路径**的测试：

```
 1 package person;
 2 
 3 import static org.junit.Assert.*;
 4 import org.junit.Test;
 5 
 6 public class PersonTest {
 7     String personName = "Bob";
 8     int personAge = 21;
 9 
10     @Test
11     public void testCreatePersonWithNameAndAge() {
12         Person p = new Person(personName, personAge);
13         assertEquals(personName, p.getName());
14         assertEquals(personAge, p.getAge());
15     }
16 }
```

第一行声明测试包的名字，下面两行引入单元测试框架 `junit`。第六行声明类 `PersonTest`，这里将包含我们的测试代码。

接着第七、八行的语句定义两个**变量**，保存姓名（`Bob`）和年龄（`21`）用于测试。为了不在测试中多处键入`"Bob"`（可能拼错）或`21`（在某些上下文中可能不能清楚表明它代表了某人的年龄），我用变量来保存这些值，并给了它们有意义的变量名。现在，如果我拼错了 `personName`，编译器会给我提示，当我引用 `personAge` 的时候，很明显就能知道这个变量指的是一个人的年龄。这种用意图明显的变量来进行测试而非硬编码的方式，称为**仪表法**，是让测试变成标准的重要技巧。

① Eclipse IDE for Java Developers, Version 4.2 (Juno) and JUnit 4.

第十行用 @Test 标注来表示下面声明的这个方法是一个测试方法。我给这个方法起了一个冗长的名称，用以表示这个方法的测试内容。第十二行初始化测试，建立一个新的 Person 类对象。注意左侧行号处有一个带着红圈的X且 Person 下面有红色波浪线，表示编译器无法识别 Person 类，因为它还没被创建。

最后两行断言，我建立了一个 Person ，他的 name 字段是 personName 的值（"Bob"），age 字段是 personAge 的值（21）。通过两种不同版本的测试方法 assertEquals() 来完成断言。第一种 assertEquals() 方法接收两个字符串或字符序列作为参数。第一个字符串代表期望的结果（"Bob"），第二个字符串来自 Person 的 name 字段。assertEquals() 比较两个字符串，验证每一个字符是否相同。如果两个字符串包含不同的字符序列，单元测试框架会输出红条。

第二种 assertEquals() 方法比较两个数。第一个参数是期望的结果（21）。第二个参数来自 Person 的 age 字段，如果和期望的结果（21）相同，则测试通过。如果 age 字段和期望结果不同，单元测试框架会输出红条。

11.2.2　存根代码

现在还没法编译测试代码，因为测试引用的符号（Person 类、getName() 方法、getAge() 方法）不存在，所以我让Eclipse创建存根。

在Eclipse中，如果我将鼠标移动到第十二行第一个 Person 实例的地方，会有一个小窗口弹出显示Person cannot be resolved to a type（未找到 Person 类型）。这表示我想要创建一个 Person 类的实例，但系统不知道 Person 是什么，因为它尚未定义。弹出窗口同时也会有一组快速修复方法，第一个就是Create class Person（创建 Person 类）。选择这个选项，Eclipse会替我创建一个存根 Person 类，就像这样：

```
public class Person {
}
```

Person 类是空的，还没有任何内容，但现在它已经存在，测试中第一个 Person 处的错误已经没了。仍然还有其他错误，在同一行的等号后，The constructor Person(String, int) is undefined（构造器 Person(String, int) 未定义）。这说明我试图用 name 和 age 来创建一个 Person，但 Person 类无法接收这些字段，因为它还没有称为**构造器**的特殊方法。再一次，点击弹出窗口中的Create constructor Person(string, int)（创建 Person(string, int) 构造器），让Eclipse来为我创建存根。这将给 Person 类添加如下构造器：

```
public Person(String string, int i) {
        // TODO自动创建构造器存根
}
```

用 name 替换 string，用 age 替换 i，让这些字段的意义更清晰，然后删除 TODO 注释。现在 Person 类变成了这样：

```
public class Person {
        public Person(String name, int age) {
        }
}
```

这个错误修复了。但测试中还有另外两个错误：getName() 和 getAge() 方法还不存在，也需要为它们创建存根。

注意，即便我只想测试创建一个有效的 Person 实例，也需要为 name 和 age 定义 getter 方法，用来“获取并返回字段信息的代码”，因为测试需要用它们获取 Person 的信息进行验证。可以利用自动生成为 getName() 创建存根：

```
public Object getName() {
        // TODO自动创建方法存根
        return null;
}
```

然后利用自动生成为 getAge() 创建存根：

```
public double getAge() {
        // TODO自动创建方法存根
        return 0;
}
```

清理存根代码，让 getName() 返回 String 而不是 Object，让 getAge() 返回 int 而不是 double。现在测试代码没有错误了，Person 类的存根如下：

```
public class Person {
        public Person(String name, int age) {
        }

        public String getName() {
                return null;
        }

        public int getAge() {
                return 0;
        }
}
```


现在可以编译了。测试中的所有引用都可以正常解析，执行测试，然后得到……红条！因为还没有 name 和 age 字段，所以返回值和期望值不同，测试将失败。

但别忘了，在TDD中我们的目标是让红条**证明我们的测试可能失败**。在证明的同时也做了其他事情——定义了一个有姓名和年龄的 Person 类，只不过尚未实现而已。

11.2.3 实现行为

接下来为姓名和年龄定义字段，在构造器中设置字段的值，然后在 getter 方法中返回。完成

之后代码成了这样：

```
public class Person {
        private String name;
        private int age;

        public Person(String name, int age) {
                this.name = name;
                this.age = age;
        }

        public String getName() {
                return name;
        }

        public int getAge() {
                return age;
        }
}
```

现在执行测试，绿条告诉我们 Person 类如期望的那样工作。现在为止，我们创建了一个有 name 和 age 的 Person 对象。

11.3　引入限制条件

如果试图将 age 设置为一个负数或一个非常大的数值会怎样？也许需要引入一些对 age 的限制。

首先为 Person 类定义一些常量，当然测试先行。

```
    @Test
    public void testConstants() {
        assertEquals(Person.MINIMUM_AGE, 1);
        assertEquals(Person.MAXIMUM_AGE, 200);
    }
}
```

MINIMUM_AGE 和 MAXIMUM_AGE 下面出现红色波浪线，因为它们尚未定义。让Eclipse在 Person 里自动生成它们，然后整理代码，我们有：

```
public static final int MINIMUM_AGE = 1;
public static final int MAXIMUM_AGE = 200;
```

用这些符号来定义年龄的最小值和最大值，从1到200。我从没听说过有人活到200岁，等到真有人活到200岁的时候，我也早已不在人世，不需要我来更新代码了。

我刚刚做的其实就是创建了潜在的遗留代码。就像编写财务软件（或者其他软件）的时候假

设表示年份的前两位数字是19。我这里依赖的是，当真的到了人类能活过200岁的时候，任何需要为此修改代码的人，都能够很容易找到名为 `MAXIMUM_AGE` 的常量，然后改成300或500或5000，因为我用了简明的语言和**见名知意**的命名——我们会在下一章详细讨论。

11.3.1 编写测试和代码存根

下一步，编写测试来验证传入的年龄不会过小。保存年龄的数据类型是 `int`，在Java里是32比特数据，可以保存–2 147 483 648到2 147 483 647的数值，所以必须防止传入负数。于是编写一个测试，当传入比 `MINIMUM_AGE` 小的数值时**抛出一个异常**。在 `PersonTest` 类中添加一个如下的测试方法。

```
@Test(expected = AgeBelowMinimumException.class)
public void testConstructorThrowsExceptionWhenAgeBelowMinimum() {
    Person p = new Person(personName, Person.MINIMUM_AGE - 1);
}
```

这个特殊的注解告诉 JUnit，当用0（`MINIMUM_AGE - 1`）作为参数调用 `setAge()` 时，不应该期望得到一个实例化的 `Person`，而是应该得到一个 `AgeBelowMinimumException` 异常，这个异常因为尚不存在所以被IDE标记出来。让Eclipse替我们生成存根。

```
public class AgeBelowMinimumException extends RuntimeException {
}
```

现在一切定义好了，执行测试，然后得到……

红条！

11.3.2 实现行为

测试已经编写好，但尚未在代码中实现行为，通过修改 `Person` 的构造器来实现：

```
public Person(String name, int age) {
        if (age < MINIMUM_AGE){
                throw new AgeBelowMinimumException();
        } else {
                this.age = age;
        }
        this.name = name;
}
```

上面的代码表明，如果用小于 `MINIMUM_AGE` 的年龄来创建 `Person` 实例的话，会抛出一个异常，而不是创建一个新的 `Person` 实例。当输入的参数没有意义的时候，应该抛出异常，而不是创建不合法的 `Person` 实例，给系统带来未知风险。

注意这里的测试是从相反方向进行的。

如果试图用小于 MINIMUM_AGE 的年龄来创建 Person 实例，我希望系统**抛出一个异常**。如果不是这样的话就是出问题了，所以我的测试验证系统在 age 小于 MINIMUM_AGE 的时候**抛出异常**，如果没有**抛出异常**的话，JUnit会显示红条。

最后，编写一个测试来验证，如果用大于 MINIMUM_AGE 的年龄创建 Person 实例，也会抛出异常。

```
@Test(expected = AgeAboveMaximumException.class)
public void testConstructorThrowsExceptionWhenAgeAboveMaximum() {
    Person p = new Person(personName, Person.MAXIMUM_AGE + 1);
}
```

用Eclipse创建异常存根：

```
public class AgeAboveMaximumException extends RuntimeException {
}
```

在实际开发中，可能会用一个异常AgeOutOfRangeException，而不是用AgeBelowMinimumException和AgeAboveMaximumException，这里是为了演示，如果有两个范围限制，我们可以分别进行不同的处理。

这时可以正常编译了。执行测试，得到一个红条，告诉我们必须在代码里处理这个新的异常。于是在 Person 的**构造器**中添加：

```
if (age > MAXIMUM_AGE) {
        throw new AgeAboveMaximumException();
}
```

11.4 我们创建了什么

在这个简单的实例中，我们创建了四个类：PersonTest，用来驱动创建 Person 类和两个异常类（AgeBelowMinimumException 和 AgeAboveMaximumException）。

这个实例的Java代码的Eclipse项目可以在 Pragmatic Programmers 网站本书页面[1]下载。

PersonTest 类如下：

PersonExample/tst/person/PersonTest.java

```
package person;

import static org.junit.Assert.*;
import org.junit.Test;

public class PersonTest {
```

① http://pragprog.com/book/dblegacy

```
        String personName = "Bob";
        int personAge = 21;

        @Test
        public void testCreatePersonWithNameAndAge() {
                Person p = new Person(personName, personAge);
                assertEquals(personName, p.getName());
                assertEquals(personAge, p.getAge());
        }

        @Test
        public void testConstants() {
                assertEquals(Person.MINIMUM_AGE, 1);
                assertEquals(Person.MAXIMUM_AGE, 200);
        }

        @Test(expected = AgeBelowMinimumException.class)
        public void testConstructorThrowsExceptionWhenAgeBelowMinimum() {
                Person p = new Person(personName, Person.MINIMUM_AGE - 1);
        }

        @Test(expected = AgeAboveMaximumException.class)
        public void testConstructorThrowsExceptionWhenAgeAboveMaximum() {
                Person p = new Person(personName, Person.MAXIMUM_AGE + 1);
        }
}
```

Person 类如下：

PersonExample/src/person/Person.java

```
package person;

public class Person {
        public static final int MINIMUM_AGE = 1;
        public static final int MAXIMUM_AGE = 200;
        private String name;
        private int age;

        public Person(String name, int age) {
                if (age < MINIMUM_AGE) {
                        throw new AgeBelowMinimumException();
                }
                if (age > MAXIMUM_AGE) {
                        throw new AgeAboveMaximumException();
                }
                this.age = age;
                this.name = name;
        }

        public String getName() {
                return name;
        }
```

```
        public int getAge() {
                return age;
        }
}
```

还有两个异常类：

PersonExample/src/person/AgeBelowMinimumException.java

```
package person;

public class AgeBelowMinimumException extends RuntimeException {
}
```

PersonExample/src/person/AgeAboveMaximumException.java

```
package person;

public class AgeAboveMaximumException extends RuntimeException {
}
```

注意，我们的测试中包含三个断言：一个是关于快乐路径的 age 为 MINIMUM_AGE，最小的合法年龄。另一个是当 age 小于 MINIMUM_AGE 的情况，第三个是 age 大于 MAXIMUM_AGE 的情况。

你也许奇怪，为什么不做第四个断言测试 age 等于 MAXIMUM_AGE 的情况。那样的测试和等于 MINIMUM_AGE 的情况是重复的，因为决定两个测试是否通过的因素是一样的。我们希望测试是**独一无二的**，所以没有那个第四个断言。

也许有的人认为有第四个断言更好。如果你这么想，我不会跟你争论。相反，只要你编写测试我就为你喝彩。

因为把测试看作标准，我们希望测试覆盖所有的代码，连次要代码也覆盖。标准中应该有的，测试集中也应该有。甚至为常量进行测试。举例来说，代码中有这样的测试：

```
@Test
        public void testConstants() {
                assertEquals(Person.MINIMUM_AGE, 1);
                assertEquals(Person.MAXIMUM_AGE, 200);
        }
```

如果有人决定 MINIMUM_AGE 应该是18并修改了代码，我们的一个断言就会失败。所有会引发行为变化的修改都应该被测试覆盖。

11.5　测试就是标准

把单元测试看作标准。这对于考虑应该编写哪些测试非常有用。

考虑如何针对线性区间 MINIMUM_AGE 到 MAXIMUM_AGE（1到200之间）的数值进行测试，这需要三个断言。

- 一个是 MINIMUM_AGE –1（也就是0），应该返回异常，因为有效值位于1到200之间，而我们传入的是0。
- 一个是区间内第一个有效值 MINIMUM_AGE（也就是1），应该没有任何异常。
- 一个是另一个非法值 MAXIMUM_AGE + 1（也就是201），应该返回异常。

注意三个断言（0、1、201）都是唯一的。它们不会因为同一原因而失败。所有测试的失败原因都应该各不相同，而且每个测试的失败原因应该只有一个。将此规则应用于代码之中。

这也是用测试集来取代需求文档的另一个巨大的优势。很难甚至几乎不可能判断需求文档是否已经过时。但是，点一下按钮就能用单元测试来验证所有的代码和测试是否同步。单元测试是**活的文档**。

对于系统中的一个功能或者行为，编写适量适度的测试进行验证。每个测试都是唯一的，而且应该通过命名进行区分。

11.6　测试需要完整

应该假设测试集是对整个用户故事的完整定义。如果没有用测试来描述一个行为，那么这个行为很可能是错的。任何没有在测试集中体现的行为都被视为不存在。所以需要用测试来覆盖系统中所有的行为。如果真正将测试先行开发付诸实践，就会有这样完整的一组测试集。永远在创建行为前编写测试。

同时，由于在编写的时候就创建了可测试的代码，我们将会看到完成后的产品在整体质量上有显著提升。代码的可测试性和质量有着密不可分的联系，但是请不要以质量保证的名义进行测试。

软件开发者和管理者很可能会说："哈，我们一石二鸟，一边进行TDD一边进行质量保证。"但如果真的这么做了，就不是在做真正的TDD了。我们会编写过多的测试。我们所编写的测试不会描述行为，而是开始曲解代码的测试标准，让其难以阅读、理解和修改。

质量保证是为了保证代码有能力处理某些特殊情况，而那些特殊情况不应该是开发者在本阶段考虑的。先把对健全性的考虑放到一边，专注于**构建行为**，然后回头根据需求整理。这样做更容易，就像在编书或写文章的时候先写完再编辑。每个作者都知道这个道理。不应该在写作的时候就自己进行编辑，那会导致写作障碍。编码也是一样。先创建行为，然后回头让代码更健壮。

但是这个观点却难以说服管理层。他们要用TDD取代质量保证。诚然，TDD会让代码更容易测试，并提高测试覆盖率，但对意外情况过多考虑，非功能性需求以及其他通常在质量保证阶段需要处理的事情依然存在。

反之，如果在最终质量保证之前关注那些潜在问题，就可以有足够的时间处理可能发现的任何问题。但不要在开发的同时进行。可以让质量工程师和软件开发者一起工作，但两件事情不能

同时占用一个大脑。在TDD中编写测试，关注对行为的描述，而将质量保证思维暂时放到一边。然后，当代码正常工作之后，回过头去用质量保证思维编写测试，尝试破坏代码，看看有什么情况会让代码出错。添加大量的质量保证测试会让构建过程变慢，所以在代码基本正常工作之后再添加质量保证测试会更高效。

11.7　让测试独一无二

优秀测试的标准是，所有失败的测试都是因为某些**已知原因**而失败的，而不会因为其他原因失败，而那个测试是**系统中唯一**可以由此原因导致失败的测试。换句话说，**测试应该是独一无二的**。

如果测试是独一无二且没有冗余的，那么在出现失败的时候就能得到明确的反馈。通常只有一个测试会失败，而不是整个测试集都失败，而且也不需要花好几周来整理测试。

很多团队实践TDD失败的一个主要原因如下。他们认为：**两个测试很好，三个更好，所以十个就一定更棒了。让我们写上一大堆测试。**

但是，最后总是会在系统中留下冗余的测试。如果是这样，在重构代码的时候，也必须重构很多的测试，这会大大拖慢进度。这意味着，如果有什么地方出错了，会有多个测试因为同一个原因失败。测试反馈变得嘈杂，难以找到真正的原因所在。

测试先行是一种**设计方法**，一定要写可测试的代码，将需求具体化，这能帮助开发者构建高质量的代码。

如果要针对某些东西编写测试，首先要找到验证的方式。所以用可控的片段来编写，片段越小越好。进一步来说，编写**专注一件事物**的代码。如果在一个类中有多个关注点，那么就会有多个原因导致测试失败。结果就是，类中的关注点越多，相应的测试数量会呈指数级增长。

11.8　用测试来覆盖代码

许多公司对单元测试的代码覆盖率有一定的要求。对我来说，只有百分之百的测试率是有意义的。

作为开发者我们并不能一直保证这个覆盖率，但我们应该努力争取，即使有时依赖于其他服务导致代码中的某些部分无法测试。

代码覆盖率工具能告诉我们，哪些代码是被单元测试覆盖的。对于何种代码覆盖率是最佳的讨论可以成为我们偷懒的借口。我见过有的公司将覆盖率定位为60%、80%或者其他的数值，而这给了开发者不编写测试的借口。他们不去测试那些难以测试的代码，而去覆盖那些简单的代码，因为它们容易测试。相反，我也见过有的开发者只测试复杂代码，而不测试简单代码，比如getter方法和setter方法。他们的理由是简单代码不太可能引入bug，所以没必要编写测试。

我力争百分之百的测试覆盖率，因为我用测试来规范代码。在我的规范中，对诸如getter方法和setter方法这样微不足道的行为也进行定义，所以也在测试中体现。因为我从来只编写让失败测试通过的代码，所以我总是达到百分之百的测试覆盖率。

如果代码有多重执行路径，意味着代码的逻辑也会更复杂，所以应该用测试来覆盖代码。每个代码执行路径得到不同的结果，所以需要针对每一个路径进行单元测试。

随着执行路径数量的增加，覆盖这些路径的单元测试数量呈指数级增长，所以TDD的好处是会让我们编写更简单的代码。

11.9　bug 是缺失的测试

每一个bug的存在都是由系统中缺失的测试导致的。在TDD中修复bug的方式是编写代表那个bug的失败测试，然后修复bug看到测试通过。这不仅仅修复了bug，而且保证以后它不会复现，因为在任何的改动引发该bug的时候都有一个测试会失败。

软件中的bug可以分为许多不同的种类。语法错误或拼写错误，还有更严重的，比如逻辑错误或设计缺陷。测试驱动开发致力于解决**所有**这些问题。

测试可以提供编译器无法提供的反馈。传统上来说，开发者在执行程序前只能得到编译器的语法检查给出的反馈，只检查源代码的语法意义而非逻辑意义。编译器只能检查语法错误和一些基础性错误。它保证代码结构正常，但并不保证代码逻辑上正确，或者做应该做的事情。

单元测试填补了语法错误和概念性错误之间的空缺。它可以捕获其他工具无法捕获的错误。它可以抓住那些我们可能忽略但会被客户发现的错误。我宁愿看到一个失败的测试，也不愿接到一个生气的客户打来的电话。我并不需要编写完美的代码——没人可以——但我宁愿这些磕磕绊绊发生在我和我的单元测试之间。

11.10　用模拟对象来测试工作流

对于指定的参数、返回结果、算法行为等，单元测试十分有用，但单元测试不能测试一系列正确顺序的调用，或其他的类似场景。为此有另外的测试类型，称为**工作流测试**。

工作流测试用所谓的**模拟对象**（mock）进行测试。模拟对象是真实对象的替代。它们只在测试环境下使用，验证待测试代码如何和外部依赖交互。代码中需要的任何外部依赖都用模拟对象替代，有许多这种类型的技术和工具。

对模拟进行设定，让它在给定一组输入的时候返回特定的结果。执行包含模拟对象的测试的时候，我们是在和模拟对象交互。问模拟对象“你是否被正确调用了？你被调用时所用的参数是否正确”有助于理解代码如何与外部依赖交互，同时不会对真实的外部世界造成影响。

11.11　建立防护网

实践TDD有助于项目构建的良好节奏，同时也给开发者编写代码提供了防护网。你可能对防护网这个概念嗤之以鼻，但是，单元测试对开发者的意义和防护网对杂技演员的意义一样重要。对于每天要做5次高空秋千表演的演员，有哪个愿意去想一个小失误都可能断送他的表演生涯甚至生命呢？防护网是精神上的保护，它给尝试新事物提供了信心，让你可以自由地进行试验，进而产生真正的创新。

进行测试先行开发诸多好处之中最大的就是这个最终结果：软件开发者编写更容易测试的代码，容易测试的代码维护的成本更低。

实践测试后行的软件开发者有时会发现，为了编写测试，他们必须修改代码让它更容易测试。如果先编写测试，一开始就可以编写出可测试的代码，这样明显更高效。实践测试先行的时候，你永远不需要重新编写代码让它更具有可测试性。

常识告诉我们，在有了可测试的东西之后再创建测试。虽然先编写测试违背了这个常识，但是它确实更有意义。同样，许多开发者需要先进行设计，但是，实际上通常是在项目的后期才会发现正确的设计。所以我提倡一个同样违反常识的实践：最后实现设计。这是下一章的主题。

11.12　让我们付诸实践

以下是把这些想法付诸实践的方式。

11.12.1　使用测试作为标准的 7 个策略

用单元测试来描述代码行为，保证开发的关注点在只构建必要的内容。这同样也有助于只编写对代码行为具有描述作用的测试。这些测试不仅成了如何使用所创建行为的文档，同样也展示了构建功能的顺序，体现了软件是如何设计出来的。以下是使用测试作为标准的7个策略。

将测试仪表化

不使用硬编码的值作为参数，而是将这些值存入具有代表性名称的变量中（比如，`maxUsers`而不是 20）。这样含义更明确，使测试能像标准文档一样阅读。

使用见名知意helper方法

将初始化行为或其他的功能代码片段放到单独的helper方法中。这让我们在创建多种不同的初始化选项同时移除测试中的冗余，同样也可以让helper方法的名称具体而有意义，且更便于阅读。

突出重点

用事物的重点命名。用名字概括事物、展示关键概念。用肯定的方式表达测试意图。不要传入硬编码的值，用具有代表性名称的变量传值（比如，`anyInt` 而不是4）。

测试行为，而不是实现

测试应该验证行为，用行为而非实现来命名。`testConstructor` 是个糟糕的名字，`testRetrievingValuesAfterConstruction` 则好些。使用长名字来准确表达测试要验证的行为。针对行为进行测试，让代码更容易测试，让开发过程更专注。

用模拟对象测试工作流

可以通过断言来测试返回值和行为，但没法测试工作流，或者对象如何与其他对象交互。如果想移除依赖单独测试我们的代码，用**模拟对象**来替代外部依赖。

避免过度描述

用测试来规范行为很容易过度描述。有时候会为了理解一个算法而编写出多余的测试。一旦找到抽象并编写出算法之后，回过头去删除冗余的测试是十分必要的，保证系统中的每个测试体现一个新的差异。

利用真实的例子

测试通过真实的行为来验证抽象需求的正确性。使用能代表系统真实使用方式的例子，这样行为会以它们在系统中被使用的方式进行测试。研究真实的例子通常会在开始编码之前揭示出一些矛盾和设计缺陷，可以尽早进行处理。

用测试先行开发构建行为的时候，会得到一组测试集作为活的文档。我们可以在任何时候执行测试，当看到绿条时就证明文档是最新的。对于重构来说，描述行为的单元测试是十分有价值的资源，也可以作为文档来展示系统的使用场景。

11.12.2 修复 bug 的 7 个策略

bug是软件产业的灾星。bug有时候难以发现而且修复成本很高。如何处理bug对于软件开发流程来说影响很大。力争零bug，至少避免囤积bug，尽早处理bug。我们可以用bug来了解优化我们流程的方法，好让类似的bug不再出现。以下是修复bug的7个策略。

一开始就避免写出bug

有个老笑话：病人举起胳膊问医生“大夫，为什么我一这样做就疼”，医生回答“那就别那么做”。我们希望从一开始就避免写出bug，并且通过编写高质量代码和利用工具避免失误来做到这一点。

尽早发现bug

如果不能避免产生bug，那么就需要有相应的流程尽早发现bug 。在bug产生和bug修复之间的周期越长，包含bug的代码对于其编写者来说就越不熟悉。但是，有了自动化的回归测试之后，开发者可以立刻知道他们的代码有没有bug。让bug从产生到修复的周期降为0，修复bug的成本也随之降为最低。

通过设计让bug更容易找到

无论你的回归测试做得多好，有些bug总会逃出我们的手掌心。如果真是这样，那么让它们变得容易查找就再好不过了。在代码中查找bug的难易和代码的质量息息相关。比如，高内聚、封装良好的软件不容易出现能引起bug的副作用。高内聚、封装良好的软件容易阅读、容易理解，从而让查找bug变得容易。

问对问题

既然开发者花费大量时间在调试上面，试图找到bug是如何产生的，迅速定位bug就十分重要。我大学的一位教授说过，试验的成败不重要，重要的是你从中学到了什么。对调试代码来说这也是个好建议。在查找bug的时候，我会尝试去构建那些可以得到bug在哪里（或者不会在哪里）的信息的场景，然后逐步缩小代码范围，直到找到问题。

把bug当作失败的测试

一旦找到一个bug，在进行修复之前我会写一个失败测试来重现bug，在修复之后测试通过。因为之前做出了错误的假设所以才出现了bug。当找到那些错误的假设，用单元测试将其收录之后，我们会得到一个覆盖那个问题的回归测试，以后就再也不用处理那个bug 了。

利用发现的缺陷修正流程

当发现一个bug的时候，我会问自己这个bug是如何产生的。有时候会发现软件开发流程中存在的问题。修正流程可以防止许多bug产生。寻找用工具帮助我们更好工作的方式。

从错误中学习

如果bug代表了开发流程中的错误假设和缺陷，仅仅修复bug是不够的，应该修复允许bug产生的环境。将bug看作是设计和流程漏洞的教训，这样就能够找到修正它们的方式。把错误当作学习的机会，从每个问题中获取宝贵信息。

bug在软件开发中消耗昂贵的成本，每年要花费数以十亿计的美元。bug不仅仅是代码中的问题，同样也是软件开发流程中的问题。如此看待bug，就可以通过它们来找到优化流程的方式，进而防止类似bug的产生。

11.13 总结

用测试描述行为，从而建立**活的标准文档**。我们展示了用测试先行的方式构建功能的例子，见识了如何用测试来描述行为。当软件开发者发现测试先行开发的益处之后，他们就会成为**测试感染者**，在一切开发工作中使用测试先行。

本章中心思想如下。

- 测试集不仅验证行为，而且描述行为。
- 通过**仪表化**测试，可以清晰理解测试意图，让测试成为**活的标准文档**。
- 测试提供了一种防护网，让我们可以重构代码，在出现错误时能立即发现。
- 通过测试描述行为，在构建行为的时候总能知道编写多少数量的测试是合适的。

如果使用得当，测试驱动开发能帮助开发者构建出可测试的代码，这样的代码更容易维护。但是，如果使用不当，测试驱动开发也会失败。把测试当作可执行的代码标准，可以知道一个行为需要编写多少测试。我们通过一个例子体会测试先行开发的基本概念，即便不是程序员也能有所体会。

第 12 章 实践8：最后实现设计

我并非在推崇将**所有的**设计都放在开发的最后阶段进行，但是有一些设计在开发周期的后期进行会更加高效且效果更好。这类设计不是在白板上进行的，而是在有了可运行的代码以及全面的测试之后进行。这是对软件进行健壮性设计的绝佳时机。

测试支持安全地整理代码，所以自然应该在测试已经到位且代码可以正常运行之后调整代码设计。项目的后期比前期有更多机会发现设计模式，可以更好地理解系统该怎么运作。

这个顺序和我们通常的顺序相反。传统的开发过程专注于让代码正常工作，然后在开发后期尽量移除bug。但是为了更好地编写出高可维护的代码，我们利用这种实践将此阶段提前到开发的前期。

让我们先编写测试，最后再进行设计。

12.1 可变性的阻碍

健壮的代码灵活且易于修改，因为它们容易阅读、容易理解，不仅是对于这些代码的开发者，对于其他专业软件开发者来说也是一样。

我们之前已经讨论过，“优秀的代码”的一个特点是容易修改。当我询问开发者是什么让代码容易修改的时候，他们通常的答案是良好的文档、见名知意的命名、遵循一致的隐喻，等等。同样也有一些其他的事情可以让代码容易理解和容易修改。

有没有一些可以让开发者遵循的指导，以便让代码容易修改？我相信答案是“绝对**有**”！我们已经讨论过一些可以让代码容易修改的原则和实践了。但是也需要问一个相反的问题，因为有时候知道该避免什么和知道追求什么同样重要。

是什么让代码难以修改？开发者做的哪些事情让代码难以在以后修改？

我相信有些被广泛接受的开发者实践实际上阻碍了可修改代码的产生。可变性的阻碍可能会成为使用代码的绊脚石。这些阻碍很多都是细节，单独来看可能都不是问题。但是当这些阻碍成为习惯，甚至实践的时候，就会严重拖慢开发节奏。

先来看看那些常见的会阻碍代码修改的开发者实践。这可能涉及技术方面，所以请非开发者忍耐一下。

以下是我列的清单。

缺乏封装

一部分代码对另外一部分“知道”得越多，则依赖越重，无论是显式的还是隐式的。这会导致微小而又难以预料的问题，进行一个小修改会破坏看似不相干的代码。只要一段代码“知道”或者依赖另外一段代码的实现，就很难在不破坏系统其他部分的情况下进行修改。

滥用继承

继承是面向对象语言中十分重要的一部分，但可能会被滥用或误用，将不相关的问题耦合起来，产生过深的继承结构，导致维护性问题。

僵化的实现

当缺乏抽象的时候，两个或多个行为之间会有太多共同性，导致冗余和非必要的复杂性，让代码难以使用。僵化的实现难以修改，难以在日后添加新的差异性。

内联代码

在受限的系统中复制粘贴内联代码，而不是将代码放入独立的方法中进行调用，被当作一个高效的实践。但是，这会让代码难以阅读，并引入冗余。时至今日，多数的编译器会优化间接的方法调用，在调用它们的方法中展开代码，这给了我们通过有意义的命名来提高代码可读性的机会，而不是在代码块前添加注释来说明。

依赖

处理依赖的方式至关重要。如果没有正确隔离依赖，可能会将原来不相关的问题耦合。将来分离这些问题会变得很困难。

使用你创建的对象，或创建你使用的对象

一开始我有些难以理解这一点，但事实上，编写可扩展的代码（那些可以用最小的代价进行扩展的代码），最需要**避免**这一点。为了实例化一个对象，需要知道很多该对象的细节，代码的使用者必须关心子类型，而这些知识破坏了封装性，让调用者更依赖某个具体实现。如果服务的使用者同时也负责服务的实例化，就产生了耦合，难以测试、扩展、复用。在本章的后面会有更详细的讨论。

12

这只是阻碍易修改代码的部分实践。这些都是小事情，偶尔发生并不为怪。但如果反复在百万行的代码中出现，就会导致大问题。

我有一个设计芯片的硬件工程师朋友。他说“软件行业太宽容”。他并不是在称赞这个行业。

在设计电路的时候，他知道一个小错误就能够破坏整个设计，从而花费成百上千美元重新制造。所以，他对自己的行为极度小心，近乎偏执，将设计交付制造之前查了又查。他认为多数的软件开发者都缺乏纪律，鲁莽行事，抓不住重点，并且随处制造麻烦。

我无力反驳他。

问题在于软件开发者（在某种程度上）不用对他们的马虎负责，大多数开发者没有被教导编写软件时要高度严谨。他们在学校里面编写的程序和将来工作中需要开发的企业级系统比起来微不足道。

对于小项目来说，马虎大意并没什么，但是当项目足够大的时候，我们累积起来的错误终究会无可避免地找上门来。

12.2　可持续性开发

为了让软件开发可持续，即让我们可以快速构建功能并且在将来很容易扩展，我们必须注重软件的可维护性。

以下是编写可持续代码的5个注意事项。

删除死代码

死代码是指那些被注释掉或不再被调用而永远得不到执行的代码，除了干扰其他开发者之外，它完全没有意义。

删了它。

如果真的需要，可以在版本库里回滚，获取那段代码，加回到现有的系统中。

保持名称更新

重命名方法和类，以保持“见名知意”的良好名称。随着开发的进行，对事物理解的加深，代码功能也会随着变化。如果代码变化了，随之也需要重新命名来反应代码更新后的行为。

集中决策

将决策集中，让决策只进行一次，保证没有冗余。如果决策需要改变，只会影响一处。这让代码修改变得更安全且更简单。

抽象

对所有的外部依赖建立并使用抽象，建立模型中缺失的实体，再次强调，模型应该自然反映建模的事物。这也让代码更容易测试、扩展、复用。

对类进行组织

谁都可能在建模领域时遗漏一些实体，这会造成难以理解系统运作。所以需要保证建模完整，然后对类进行组织，让它们有合理的行为和属性。

12.3 编码与清理

编写软件需要多种不同的思维活动。软件开发者必须在头脑中跟踪许多事情。编写软件需要关注许多细节，需要大量训练。

这绝非易事！

开发者需要注意很多事情，而且必须思考到比日常所需的更细枝末节。与此同时，还需要进行抽象，需要进行实现，需要进行设计——日复一日。

将这些不同的思维活动分不同阶段进行，这样有助于开发者保持思路清晰，但不仅仅如此。通过这种方式得出的设计更加优秀，更有弹性，更能反映问题的本质，所以也更容易理解和维护。

将编码和清理分开，当作不同的任务，我发现这非常有帮助。编码的时候，我们寻找特定问题的解决方案。清理的时候，我们将已经可以工作的代码变得更健壮。将这两者分开，是因为把它们当作不同的任务执行更容易一些。只关注实现行为，让测试通过，这样编码更容易。有了可以正常工作的代码，并有测试支持，进行清理也更容易，可以更专注于代码的易读性和易用性。

留意那些优秀的编程实践。只要不停地对代码做小改进，最终会得到一个良性循环，而不是导致技术债积累的恶性循环。

在小范围内（测试先行开发的重构阶段）和大范围内（周期性的重构工作）偿还技术债来保障团队对代码的掌控。

有时候技术债会随着时间的延长越积累越多。为了添加新功能，或者找到更好的方式，必须修改设计。这些事情不可能在测试驱动开发的重构阶段全部完成。这种情况下通常每几个月做一次大规模的技术还债。通过大规模的重构还清大块的技术债。单元测试会提供必要的帮助。

12.4 软件被阅读的次数比编写次数多

代码就像新闻一样会被阅读，也许有些人会觉得这很奇怪。通常软件被阅读的次数是编写次数的十倍。

为了效率，也为了扩展性，软件应该容易阅读。为读者（他人）编写代码，而不是为作者（自己）编写代码。软件开发不是一个一蹴而就的工作。软件开发是一个需要千锤百炼的行当，代码需要不断加强、清理、优化。

为了保证持续满足用户需求的变化，需要让代码灵活、可变、易用。我们知道应该用“见名知意”的命名取代注释来描述代码。应该用注释来描述**为什么**要做这些事而不是做**什么**。代码本身应该说明自己在做**什么**。

当然，有一些例外情况，比如大量并不按照文档描述的那样工作的应用程序接口（Application Programming Interface，API）。有些 Windows API 的工作方式和它们的文档并不一致。所以我会添加一个注释：“应该这样调用，但我发现并不生效，所以我那样调用，它生效了。”

这样我就在系统中共享了知识，这是好事。如果开发者因为担心阅读者不能从代码中理解其意图而添加注释，那么就应该重新编写代码让意图更加明显。

12.5　意图导向编程

如果我要写一个公用的应用程序接口、方法，或者暴露给外部的其他服务，我不会在那个方法中放入任何实现。我会简单地将其委托给其他的方法。

我这样做是因为这会让代码更容易阅读。如果委托一部分代码，实现一部分代码，这是视角转换，有时在概念层面处理问题，有时在实现层面处理问题，就如同在脑海中有一个任务开关在来回切换。虽然看起来不麻烦，但不停地切换会非常累人。

将所有公用的应用程序接口中的代码都委托到不同的方法中，就可以消除这些重复工作。这样代码读起来就像一段脚本或一个菜单，因为它保持同样抽象层次。**它做这个，做那个，然后做那个……很容易理解**。

我们将这个技巧称作**意图导向编程**，它让代码具有视角的内聚性，也就是说，所有的代码都在同一个抽象层次上，所以更容易阅读和理解。

如果其中一个步骤出现bug，我们能准确知道去哪里查找问题。并不需要在大量代码中摸索，因为我们将逻辑（每一小点逻辑）都进行了合理抽象。

通过不同层次来看待面向对象的代码。这是我们自然的思考方式。如果思考的是今天要处理的高级事物，我们就不会思考一些细节。然后，在需要的时候再将高层次展开，对细节进行研究。通过同样的抽象层次理解和研究代码。这就又回到了“做什么”和“怎么做”的问题上。如果要写一个待办列表，应该写下需要做**什么**事情，而不是（至少不是详细地）写下要**怎么**做每件事。

如果关注于“怎么做”，深入到具体“做什么”的细节，我们会发现在实现这个“怎么做”的时候还需要更多的“做什么”。先思考“做什么”，将“怎么做”委托给其他部分，直到深入最后的细节实现部分。

这听上去像是个骗局，一开始我也这么觉得。我觉得“面向对象”就是忽悠人的。但是事实上，在每个抽象层次上，我都对代码有更好的理解。这种理解需要在代码中体现。

代码就像是思考，尽管有时候我觉得开发者抗拒这样的想法。我们不想劳神去那样整理代码。但是那样的确有帮助，帮助我们清晰地思考。

如果待办列表上有一项“到银行还车贷”，那是一项“做什么”，是我计划要做的事情。我不会写下“拿到车钥匙，穿上鞋，坐进车里，发动引擎，……”。到银行还贷的具体流程会变得非常复杂，如果想要完成还贷，我需要将这一切都完成。

但真的需要都写下来吗？

我要做的其中一项是“开车到银行”，我该怎么做？另一项是“写支票”，又该怎么做？可以将任务无限展开，但是需要展开到的最细程度是什么？

从编程角度来说，我们可以展开到的最小粒度是逻辑门：“与”“或”“非”。一切都可以分解为逻辑门，我们能通过**抽象**将各种逻辑进行分类聚集。

这是科学、有效、强大的思考方式。

12.6 降低圈复杂度

1976年12月，Thomas J. McCabe在他的论文《复杂性度量》[1]中最早提出圈复杂度的概念。它代表代码中的路径数量。

只有一个条件（`if`语句）的代码的圈复杂度是2，表示代码中有两种可能的路径，也就是说代码可以产生两种不同的行为。如果代码中没有`if`语句，没有条件逻辑，代码的圈复杂度则是1，代码只会产生一种可能的行为。但是这种增长是指数级的。两个`if`语句的圈复杂度是4，三个`if`语句的圈复杂度是8，以此类推。尽你所能降低代码的圈复杂度，因为一般来说，一个方法所需的最少测试数量等于其圈复杂度。

不可能总将圈复杂度降为1，因为那样代码中就**没有**条件语句了。代码中没有分支，也不进行任何考虑。事实上，从某种意义上来说，计算机的定义就是一个不可能在没有条件语句情况下运行的系统。但是，方法中的**条件语句越少越好**。

条件逻辑代价高昂。如果我们研究一下编程语言中各个关键字的维护成本，就会发现每一种语言中的差不多30左右的关键字的维护成本并不相同。控制流程的关键字成本高很多：**if**、**switch**、**一元处理**、**循环**、**异常**……，所有这些条件语句都会改变控制流程。这些分岔让大脑处理起来费时费力，有两条代码路径而不是一条，这会分散我们的注意力。

举例来说，如果代码中有一个`if`语句，就必须考虑并测试真假两种情况。这意味着要在大脑中存放两种情况，这会变得有些困难。我曾经看到一个客户的代码有连续6个条件判断，也就

① McCabe, Thomas J.“A Complexity Measure.” *IEEE Transactions on Software Engineering* Volume SE-2, no. 4 (1976) http://www.literateprogramming.com/mccabe.pdf

是说圈复杂度为64。我的大脑远远处理不了这么多的条件判断。多数人都不能，这就是为什么圈复杂度越高的代码出bug的概率也越高。

如果用低圈复杂度来构建每一个实体，就需要更少的测试来覆盖代码。我倾向在代码中使用多态这样的技术，将条件语句放在对象的创建阶段而不是使用阶段。将这两个阶段分离是让代码解耦、容易使用、容易维护的重要方法之一。

12.7　将创建和使用分离

在面向对象编程中，有对象的创建阶段（称为**实例化**）和对象的**使用**阶段。将这两个阶段分离，让一部分对象只负责创建对象，另一部分对象只负责使用对象，这是非常有意义的。创建对象的对象叫作“工厂”。工厂对 new 关键字进行了封装。new 关键字是用定义好的类型来创建可执行对象的方法。new 有一个条件，就是它需要知道要创建的是什么。

如果我说：“我想让你为我做点事。”

你问：“什么事？”

然后我说：“某些事。现在做吧。”

估计你会笑我的。

不要在代码中这么做。在代码中，需要有**创建**对象的能力。在用 new 创建一个对象的时候，需要传入想要创建对象的类型。这是必须的，就如同我想让你帮我做事，就必须告诉你我想要你做什么，否则你不知道该做什么。如果不知道要创建什么，就不是 new 的问题了。

然而，面向对象中有一个被称作“多态”的技术。听起来复杂，但概念很简单：可以在不知道实现**细节**的情况下**进行**工作。

举例来说，我想将一个文件用压缩它的软件来解压缩。如果是zip文件，就用unzip。如果是pack文件，就用unpack。我不关心压缩方式的差异。我只知道我有个被压缩的文件，我想用任何可以做到的方式解压它。

就好比我告诉你“帮我做些事”又不说具体做什么，这对计算机来说是困难的。计算机需要知道**具体的**解压缩方式。所以我让系统的其他部分去处理。在这个例子中，负责判断使用哪个解压软件的对象知道文件是如何压缩的。不应该让其他程序中的客户端代码去判断具体的压缩方式。客户端代码应该简单地说“解压吧”，然后相应的解压软件就解压文件，如同魔法一般……但并非魔法。

这就是所谓的多态。我们可以构建相互独立的代码，所有代码各司其职。举例来说，如果出现了一种新的之前没有的压缩软件，现有的代码应该可以自动适用，因为选择压缩软件并不是它的职责。代码将职责委托给了压缩软件。只要压缩软件正常工作，所有代码就都能正常工作。这种技术让我们的代码和系统中其他的代码保持独立，允许我们安全高效地扩展现有系统。

为了达到这个目的，我们必须将对象的创建和对象的使用分离。将对象的创建过程隔离，就可以将具体使用哪个对象的知识对系统的其他部分隐藏起来。

这样做不仅可以解耦业务规则和行为，也可以解耦调用者和服务之间的依赖。用模拟对象替换这些服务，将代码和服务隔离，让代码可以单独进行测试。

将对象的创建和使用分离，可以让你打破依赖，增进可测试性。这会让代码解耦，更容易修改和维护，也会简化代码，因为工厂创建对象但不使用对象，所以更容易测试：传入业务规则，然后观察工厂返回的对象。这种方法也会让我们使用代码更简单，因为条件语句更少，且无需关心对象的实例化。

当然，如果需要使用编程语言或者框架的功能，也需要显式创建对象。但是对于也许会在将来进行扩展的类，或需要用模拟进行测试的外部依赖，我会保证用其他的实体进行实例化，并通过一个服务对象传给其他对象使用。

12.8 演化式设计

迭代开发自然产生演化式设计，我们先由单一的可测试的行为开始，然后持续改善，直到演化出我们的设计。用测试先行开发增量式地构建软件，同时注意设计原则、开发实践和技术债，这比试图在一开始搞清楚一切更容易产生优秀的设计。

我们在开发软件时注意到的那些挑战，其实就是在我们耳边悄声说：“嘿，这里有更好的处理方式。嘿，这里有更好的选择。”我们就可以将这些糟糕的事情——bug、痛苦、客户所愿不遂的抱怨，所有我们不希望看到的可怕事物都转变成资源。

它们是关于如何更好地进行开发的线索。如果你将这些信息利用好，它们就会变成苦口良药。有针对地处理问题比凭空的预期更容易。新需求来了，想清楚如何实现这些需求，然后开始构建。我们称之为**即时性**设计。它可以避免过度开发，因为我们不需要进行预期，只需要构建所需的。这可以让需求随着开发而演变。注意问题带来的推动力，它会帮助我们更深刻地理解问题，从根源上做出更优秀的设计。

为了实现演化式设计，必须注意代码质量和可测试性，注意实施优秀的原则与实践。如果我们编写糟糕的代码，在下次迭代再面对那些代码的时候，就会发现越来越难以使用。也许不会在几周内就必须推倒重来，但我知道很多团队花了四到五年的时间，结果构建出的系统完全不可维护，只能哀叹：“如何是好？还是从头来过吧。”

没有人想陷入这般境地。

12.9 让我们付诸实践

以下是把这些想法付诸实践的方式。

12.9.1　进行演化式设计的 7 个策略

演化式设计，有时称为即时设计，是一种非常先进的软件开发方式。如果使用得当，可以成为一种效率极高的开发方式。但它并不是一个新手就能掌握的技术，需要对许多领域都有深层次的理解。以下是帮助你掌握演化式设计的7个策略。

理解面向对象设计

使用面向对象的语言并不能让软件变得面向对象。多数用 `class` 语句花括弧编写的软件是面向过程的。优秀的面向对象代码是通过封装良好的实体构建的，这些实体对要解决的问题进行了准确建模。

理解设计模式

设计模式是管理复杂度和隔离行为的好工具，可以在不影响系统其余部分的前提下增加变数。实践演化式设计时，模式比预先设计更有效。在构建软件的过程中，我们会发现很多使用设计模式的机会。

理解测试驱动开发

自动化回归测试集可以作为防护网来支持对系统的修改，测试驱动开发使用得当的话，还有助于使用优秀的设计原则和实践。

理解重构

重构是在不改变外部行为的前提下修改设计的过程。它为微观和宏观上的代码调整提供了绝佳的机会。我大部分的设计都是在已经有了可以工作的代码后进行重构的，这让我可以关注于代码的优化，产生优秀的设计。

关注代码质量

代码质量是所有的优秀软件的基础。不具备CLEAN特性（内聚、松散耦合、封装良好、自主和非冗余）的代码，很容易演变成没人愿意碰的遗留代码。关注代码质量会展示出更好的构建可维护软件的方式，让我们的设计更强壮，代码更容易维护。

要冷酷无情

即使在不清楚所有状况的时候，也很容易依赖某种设计。了解一个设计的局限并愿意对其进行修改是进行演化式设计最重要的技巧。

培养优秀的开发习惯

对于设计来说，敏捷实践中的Scrum和极限编程是十分有价值的工具，但工具不会进行设计。为了创建优秀的设计，首先要理解这些实践背后的原则，然后让优秀的开发实践成为习惯。这样，在日常工作中就可以从中受益。

演化式设计需要我们在构建软件中知道自己有多少选择，避免将自己逼入墙角。理解并使用这些优秀的开发实践，我们就有了修改设计的能力，有信心用它应对将来的变化。这让开发过程压力更少而且更有趣。

12.9.2 清理代码的 7 个策略

现在我们有时间并得到管理层的认可来清理代码。该如何清理？对遗留代码重构就像解开绳结一样，一开始可能会无从下手。以下是清理代码的7个策略。

让代码自我表达

用见名知意的命名来编写代码，让代码的意图变得明显。让代码自我表达，避免对代码的行为进行过度注释。看到用很多注释解释的代码，我就会觉得它的开发者担心我只阅读代码难以理解。

为添加测试创造间隙

对遗留代码所做的最有价值的事情是添加测试用以支持将来的重构。通常，遗留代码纠缠不清难以测试。Michael Feathers在《修改代码的艺术》[Fea04]中分享了一系列技巧，在遗留代码中添加间隙让遗留代码更容易测试。这些技巧让软件更独立且更容易测试。

让方法更内聚

最重要的两个重构方法也许就是“提炼方法”和“提炼类”。方法通常会过重。其他的方法（有时是整个类）会在冗长的方法中迷失。通过能命名的最小功能点来提炼新方法，将长方法缩短。Bob Martin大叔说，理想的方法长度不应该超过四行代码。虽然听上去有些极端，但用一个方法名来描述你做的事情，也是将代码分散成小方法的好策略。

让类更内聚

遗留代码的另一个问题是类常常责任过多，这使得它们难以命名。大的类成为多个问题的耦合点，使得它们之间的联系比应有的更紧密。用类来隐藏其他的类，让那些类职责过多难以修改。应将类打散，让它们更容易阅读和使用，让设计更容易理解。

集中决策

如果类和方法都非常内聚，业务规则就能分散到系统的各个部分，让其难以理解和修改。尝试将各种流程的规则集中化。尽可能将业务规则提炼到工厂中。如果决策集中了，也就消除了冗余，让代码更容易理解和维护。

12

引入多态

在想要隐藏多种多样的行为时引入多态。举例来说，我们有多种执行任务的方式，比如排序文档或者压缩文件。如果不希望调用者关心它们使用时候的差异性，可以使用多态。这使得

在添加新的差异性的时候，现有的客户端代码无需修改。

封装构建过程

多态的重要一步是客户端通过基类来调用派生类。客户端调用 `sort()` 而不关心具体使用的排序方式。因为要对客户端隐藏排序方式，所以客户端不能实例化对象。通过静态方法让对象负责创建自己，或者将创建的职责委托给工厂。

重构代码是开发中必要的一部分。重构会降低添加新功能时的成本。重构就是事后诸葛亮，在事后有更好的认识，帮助清理设计，让代码更容易维护。

12.10　总结

最后实现设计，通过意图导向编程来降低复杂度并提高可变性。

本章中心思想如下。

- 我们无法确保质量，只能追求质量。所以，与其关注如何验证质量，不如关注如何构建高质量。
- 容易阅读和理解的代码是可伸缩的，更容易修改（因此也更划算）。
- 意图导向编程会产生内聚的视角：所有代码都在同一个抽象层次，进而更容易阅读和理解。
- 通过分离对象的创建和使用，来提高可测试性和移除依赖。
- 演化式设计不适用于新手，它需要严格注重代码质量和可测试性。

关注可维护性并对设计进行调整，会让设计体现我们在开发中学到的东西，将我们从遗留代码的死亡漩涡中拯救出来，让代码在演进中持续优化，变得容易使用，降低维护成本。

第 13 章

实践9：重构遗留代码

重构是指在不改变外部行为的前提下对代码的内部结构进行重组或重新包装。

想象一下，如果你是若干年前的我，正在对经理说你要让整个团队花上两周（一个完整的迭代周期）来重构代码。经理问："好的。你会给我什么样的新功能呢？"

我说："等等。我是说**重构**。重构修改内部结构而不改变外部行为。**不会**有任何新功能。"

他看着我问道："那你为什么要重构？"

我应该如何回答？

软件开发者时常遇到这样的情况。有时候不知如何作答，是因为我们和管理层存在沟通障碍。我们使用的是开发者的语言。

我不能告诉经理重构代码是为了好玩，是因为它让我感觉良好，或者因为我想要学习Clojure或者其他新技术……这些对管理者来说都是不可接受的答案。我必须强调重构对于公司的重要意义，而且它确实意义重大。

开发者知道这些，但需要用恰当的词汇也就是商务用语来表达，其实就是**收益**和**风险**。

我们如何在降低风险的同时提高收益？

软件本身的特点决定了其高风险和多变性。重构能降低以下四个方面的成本：

- 日后对代码的理解
- 添加单元测试
- 容纳新功能
- 日后的重构

很显然，如果需要添加新功能或修复bug，就应该重构代码，这很容易理解。如果你以后不再碰代码，也许就不需要重构了。

重构是学习新系统运作机理的有效方式。通过对见名知意的方法进行封装或重新命名来学习代码。通过重构，我们可以补充之前缺失的实体，改善之前编写的糟糕代码。

我们都希望看到进度并如期交付，所以有时会做出妥协。重构代码能清理之前造成的障碍，为的是最终能够有所成果。

13.1　投资还是借贷

我在2009年4月的博客中首次讲述了以下故事。[①]

> 有些经营者认为开发软件是种一蹴而就的活动。如果软件在编写完成后不会变更的话确实如此，但我们身处的世界在变化，不变的软件很快就过时了。
>
> 代码的衰变是真实存在的，即使一开始编写良好的软件也常常难以预计将来会面临的变更。这实际上是好事！不需要变更的软件通常是没人使用的软件。我们希望自己构建的软件为人所用，为了软件能持续给人带来价值，它需要容易修改。
>
> 我们可以用纸板建造一间漂亮的房子，在晴朗的夏日它能支撑得住，但第一场暴雨就能摧毁它。建筑工人有一系列严格的标准和实践用来保证建筑的稳固。软件开发者也应该这样做。
>
> 我在东海岸的一位客户是全球最顶尖的财务公司之一，他们饱受大量遗留代码的摧残。大部分的遗留代码都是由合同工开发的，有些虽然由他们中的顶尖开发者开发，但是在第一版本完成后，这些开发者立刻加入了其他项目。一些初级开发者被指派来维护这些系统，有些人并不理解最初的设计，所以对系统强加修改。最后弄得一团糟。
>
> 在一次他们的高级经理及高级开发者参加的会议上，我说："我这样说对不对？你们之所以成为顶尖的财务公司，是因为你们找到一流的基金经理管理你们的基金，权衡最佳投资，然后冻结那些资产，将这些基金经理撤出去做其他项目。"
>
> 他们说我前半句说得没错，但是基金经理一直管理着基金，持续对资产做出调整，因为市场瞬息万变。
>
> "哦，"我说，"所以你们雇用一流的软件开发者，让他们进行设计，开发系统，在他们完成后就换到其他项目中。"
>
> "你的意思是我们的软件也是一种关键资产，和其他资产一样需要维护？"一个经理问道。
>
> 答对了。

你会买一辆9万美元的奔驰车，然后因为嫌花费太多而不将其送去保养吗？不会。无论车有多昂贵，制造工艺有多精良，都是需要维护的。没有人会在盖房子的时候想着永远也不会更换地毯，添置新厨房器具或重新粉刷。反之，也没有人会在开车上路三天后就换变速器，理由是早晚也得换？如果有天发现倒车档失灵，你会在这种无法倒车的情况下开多久再去检修变速器呢？

---

① Bernstein, David Scott. *To Be Agile* (blog). "Is Your Legacy Code Rotting?" http://tobeagile.com/2009/04/27/is-your-legacy-code-rotting

有些事情最好放到最后处理，有些则不能推后。知道这两者之间的差别绝对重要。

对于那些会不断累积的技术债，尽快偿清几乎总是（肯定会有例外）正确的选择。如果任由技术债在系统中堆积，而开发者又在系统中工作，那么绝对会发生冲突。开发者会碰到那些技术债并且一次次付出代价。他们没法倒车，所以必须调整他们的行为（驾驶习惯），绕弯路到达目的地，为的是不使用倒车。一个问题会导致更多的问题。越早处理技术债，花费的成本就越低，就像信用卡欠款一样。

13.2　变成“铁公鸡”

技术债和财务债一样：利息会把你拖垮。

我曾经和财务信贷公司合作过，他们有一个不愿意示人的词语用来形容我这类人。我是那种总是在收到账单时就全额还款的人，从来不让我的账上产生利息。信用卡公司称我这样的人是“铁公鸡”，因为无法从我们这样的人身上挣到钱而讨厌我们。他们喜欢那些让债务堆积每次只偿还最小还款额的人。我认识的一个人欠了一家信用卡公司1.7万美元的债。如果他每个月只还最小还款额的话，需要花93年共计18.4万美元才能还清。

和财务债一样，无视问题并不能让问题消失。我希望你成为技术债的“铁公鸡”。

有时候，我们必须让技术债多存活一阵子，要不就是现在不是修复的最佳时机，要不就是我们不知道如何下手，或者单纯的没有足够时间而已。我们会经常无可奈何地发现自己处于这番境地。但是，先支付几个月的最小还款额来度过难关，然后再连本带利一起还清，和装作相安无事直到二十二世纪，这有很大差别。

我们并非试图创建完美的代码。我始终在强调这一点。没人可以做到完美无缺，软件开发者也不是在追求完美无缺。我们必须时刻清楚地权衡利弊。我是否时不时在代码中引入了问题？是的，无可避免。如果不这样，我就会丢掉饭碗。

13.3　当代码需要修改时

即使是写得最糟糕的遗留代码，如果我们不碰它的话也能持续产生价值——只要不进行修改。

这种判断应该分不同情况讨论。任务关键型软件的需求和视频游戏完全不同。软件之于实物机械的一个好处就是，信息不会磨损。但是，遗留代码需要修改和扩展的时候会怎样呢?

如果软件真的被使用了，人们就会发现更好的使用方式，然后提出修改需求。如果想让用户从你创建的软件中获得更多价值，就需要找到安全的方式来改进代码，以支持变化的需求。

既然我们已经知道优质代码的一些特性，就可以用重构这个工具来安全地、渐进地将代码变得更容易维护和扩展。对设计糟糕的代码进行安全地重构，让我们可以注入模拟对象来使软件可

测试，这样就可以在代码中添加单元测试。单元测试这张防护网可以支持我们为了安全地添加新功能而进行更复杂的重构。

这样清理遗留代码可以在遗留代码之上工作而又不用担心引发新bug：进行渐进式修改，添加测试，**然后**添加新功能。如果有良好的单元测试来覆盖代码，就可以在绿条之上进行新的开发和重构了。这是更安全也更廉价的修改软件的方式。

在软件行业中，有许多的代码——遗留代码——并未按照我们预期的那样运作良好，完全没法维护，更不用说扩展了。但是我们能做些什么？又应该做些什么呢？

多数的情况是，什么也做不了。

在软件行业，我们不应该将遗留代码视为**定时炸弹**，而是**地雷**。如果代码正常工作，不需要修改或升级，就不要动它。这适用于绝大多数遗留代码。正如有句谚语所说："东西没坏，就别去修。"

如果我们乱碰那些遗留代码，一定会出问题。如果代码如预期的那样运行，就这样使用。这对于大多数现存的软件来说都是正确的。一般来说，只重构需要修改的代码。

如果你想修复代码中的bug，或者添加新功能，或者修改现有功能，对已有代码进行修改就非常有必要。修改代码的风险和成本都很高，所以要谨慎行事。但是，如果真的需要修改代码，就应该使用正确的方法，好让修改变得安全。事实上，这些修改代码的技巧和我们之前讨论的编写优质代码的技巧是一样的。

我们可以用重构新代码的方式重构已有代码。

13.3.1　对已有代码添加测试

测试先行能让代码更容易测试，后期添加测试比测试先行更有挑战，却也能从整体上提高代码的可维护性，降低修改代码的成本。

变更请求是好事，意味着有人关心并希望代码得到改善。

得到变更请求之后，我们希望能够做出响应，在已有的软件中提供新功能，让客户能够在使用过程中得到更多价值。当然，还有许许多多的代码已经无人问津。那些代码可以安静地被淘汰，但是，那些正在使用着的比特——客户所倚仗的很可能会变化的那些比特——是我们重构的目标。

13.3.2　通过重构糟糕代码来培养良好习惯

重构是一个未被所有开发者都理解的技巧，重构也是培养良好开发习惯、展示构建可维护代码方法的工具。这些技巧自始至终是软件工程师应必备的技巧。

重构遗留代码听上去很无聊，但事实上充满惊奇和挑战。不断练习会让人得心应手。精通重

构之后，有趣的事情就会发生：我们不再会写出糟糕的代码或遵循有缺陷的开发实践，而开始**前构**（参见《前构》[Pug05]），也就是说可以一开始就写出优质的代码。学习如何避免软件开发中的错误，如何正确进行开发，重构是我所知的最快速的方式之一。

13.3.3　推迟那些不可避免的

身为软件开发者，我们的目标是通过构建有价值的软件来创造价值。这意味着我们开发出来的软件需要能立刻产生价值，并在以后的日子里持续产生价值。

为了让软件在将来持续产生价值，必须降低软件所有者的开销，这样对软件进行改进和扩展才会有收益。让软件健壮且可维护是我们的首要目标，这样会降低软件所有者的开销。

但无论早晚，软件都会被淘汰。有些我编写的软件存活的时间让我大吃一惊。我在孩童时代编写的并不引以为傲的代码却以某种形式存活至今。

软件有时会夭折，有时也会比我们预期的存活得更长久，我们在编写软件的时候完全没法准确预估将来到底会怎样。但是我们都希望自己的软件能够持续产生价值。应该尽我们所能，提高对投资的回报，降低软件所有者的开销。

13.4　重构技巧

在我们清理代码让其更容易测试的时候，也让它变得更容易扩展且降低日后修改的开销。以下是一些重构代码的技巧。

一般来说，重构遗留代码从功能层次开始，因此可以根据一些可观测的行为编写图钉测试。

13.4.1　图钉测试

图钉测试是非常粗粒度的测试，它测试的可能是成百上千行代码的单一行为。虽然最终期望的是许多更细粒度的测试，但是，通过图钉测试来覆盖整体行为会让我们有一个落脚点。每次修改代码后，都可以回到图钉测试来验证最终点对点的行为是否依然正确。

由于图钉测试粒度很粗，因此必须频繁执行来验证修改是否对代码造成影响。这可以给一些相对安全的重构行为提供防护网，让代码中有更多的间隙可以进行依赖注入。这将有效解耦对象和它们所使用的服务，让我们可以用模拟对象替代服务，以便独立测试指定代码。这让更小单元的行为变得可测试，可以添加更细粒度的单元测试，用来支持更复杂的重构。

这就是重构遗留代码的方式，一点一点，进行小规模增量优化。遗留代码的产生是因为一直以来开发者所做的小修改降低了代码质量。修复的方式也是用小规模的代码修改来增进代码质量，然后逐渐减轻遗留代码的负担。

13.4.2　依赖注入

我们在之前讨论过分离对象的创建和使用的意义。这是让代码变得可测试的重要环节，同样也让代码变得可以独立部署。分离了对象的构建和使用，我们就可以在不引入耦合的情况下注入所需的依赖。这是面向对象编程的基本技巧之一。

各种框架都会使用这种技术，像Pivotal Software的开源Spring和Red Hat的Hibernate，都被称为**依赖注入框架**[①]。原理很简单：我们不直接创建要使用的对象，而是让框架替我们将对象注入到代码中。[②③]

用依赖注入取代创建可以解耦对象和它们所使用的服务，这会让软件更容易测试和扩展。如果不注入真实的依赖而是注入模拟依赖，就很容易测试代码。依赖注入也会使代码更容易维护，它有助于将业务决策集中化，简化对象使用方式。通常，为了理解一个新系统，首先看的就是对象实例化的地方。我们查看工厂对象，依赖注入框架，或者其他进行对象实例化的地方。这会告诉我们很多关于系统的信息，让我们知道如何改进它。

13.4.3　系统扼杀

如果需要在不影响系统的前提下替换一个组件，通常使用Martin Fowler在2004年最早提出的**系统扼杀**[④]。先用一个自己的服务将原来的服务包裹起来，然后一点点替换原来的服务，直到原来的服务最终被**扼杀**。

为新的服务创建一个新的接口来取代老的服务。然后客户端代码使用新的接口，即使新的接口仅仅是指向老的服务。这样可以阻止老的服务继续扩散使用，让新的客户端使用新的接口，新的接口背后的代码最终会被替换为新的整洁的代码。

这样就可以不慌不忙地重构已有的系统，直到老的接口仅仅是薄薄一层外壳，它只是对重构好的新代码进行调用而已。可以选择继续支持遗留的客户端，或者让它们也进行重构，用全新的接口彻底淘汰遗留系统。系统扼杀是非常有效的重构遗留代码的方法，它可以在重构的同时保持系统持续可用。

13.4.4　抽象分支

这里要介绍的最后一个技术也是Martin Fowler提出的，它叫**抽象分支**[⑤]。先不管名字，这是一个版本管理技巧，帮助我们**消除**分支。原理是针对想要修改的代码提取出一个接口，然后编写

---

① 准确地说，Spring是依赖注入框架，而Hibernate是使用了依赖注入的ORM框架。——译者注

② Spring. http://spring.io

③ Hibernate. http://hibernate.org

④ Fowler, Martin. *Martin Fowler* (blog). “Strangler Application.” June 2004. http://www.martinfowler.com/bliki/StranglerApplication.html

⑤ Fowler, Martin. *Martin Fowler* (blog). “Branch by Abstraction.” January 2014. http://martinfowler.com/bliki/BranchByAbstraction.html

新的实现，但是老的实现依然参与构建，在构建期间用功能开关隐藏正在开发的功能不让用户感知到。

一切就绪之后，可以打开功能开关用新的接口替代老的接口。这是简单且直观的方法，却消除了软件的版本分支依赖。版本分支依赖对于很多软件开发团队来说是个大问题。正如之前所说，一旦开始用功能分支，我们就不再是进行迭代开发，而是沦为了瀑布式开发。

有时系统耦合之紧密，无法在不影响其他地方的情况下打破依赖。这时Ola Ellnestam和Daniel Brolund的《天皇法则》（*The Mikado Method*）[BE12]能帮助我们解决这种纠缠不清的依赖。

13.5　以支持修改为目的重构

当然，还有许多其他应对遗留代码的技巧。基本原理都是：清理代码，让代码容易维护、容易理解，然后添加测试以便安全地进行修改。最后，必须在有单元测试的保障之后，对代码进行大规模重构。

我们把重构当作一门学问，而且还有很广泛的研究空间。我们需要继续研究这种规范化的代码修改方式。我更愿意有一套安全且可复用的修改代码方法论来分享给其他开发者，而不是凭着直觉修改代码（这种直觉无法直接告诉其他人），尽管有时候这样做效率更高。

重构软件的目的就是可以容易地根据客户的希望修改软件。这无法通过阅读客户的思想或预知未来来达成，而是遵循着一些健壮性的原则和实践，让代码在需要的时候可以被修改。这需要有一套单元测试、准确的领域建模、合理的抽象、CLEAN的代码以及其他优秀的技术实践。当我们做到这些的时候，会让修改代码变得无痛，可以随时修改代码，更好地迎合客户的需求。

13.6　以开闭原则为目的重构

这是改变我一生的几个字，它帮助我摆脱了遗留代码的窠臼。重构是在不改变外部行为的前提下调整设计。**开闭原则**是指软件实体应该“对扩展开放而对修改关闭”。换句话说，力求在添加新功能的时候做到添加新代码并将现有代码的修改最小化。避免修改现有代码是因为很可能会引发新的bug。

以开闭为目标重构是安全高效添加新功能的方式。让每一个改动都分为两步。第一步重构想要扩展的代码，让它可以容纳新功能。这并非添加功能，而是在原有的软件中通过添加抽象或定义接口之类的方式来给新功能创建空间。在有单元测试的前提下重构代码来容纳新功能，这样做是安全的、没有阻碍的，如果你犯了错单元测试立马就会告诉你。这是安全且代价小的修改代码的方式。

当代码重构完成可以容纳新功能的时候，第一步重构阶段就完成了。接下来的一步是增强阶段。先编写失败测试描述要实现的新功能，然后添加功能让测试通过，以开放的增量方式开发。

我们只添加代码，因为之前的重构阶段已经完成了修改代码。我们可以在不修改过多代码的前提下添加代码，这样更安全。最后，重构新添加的代码，让它容易理解和维护。

如果尝试一次将所有的事情都做完，正如很多开发者做的那样，那很容易迷失其中然后犯错误，结果为之付出巨大代价。但是，在有单元测试覆盖的前提下分阶段做，会让修改软件变得更简单、风险更小。

我总是将修改代码分为两个步骤，这样做是因为我通过TDD来构建功能。需要在现有系统中添加功能的时候我也这样做。前面讨论的许多实践不仅对编写新代码适用，对遗留代码也同样适用。一旦理解了优质代码是什么样的，就更容易认识到重构遗留代码的时候应该追求什么。

13.7　以提高可修改性为目的重构

代码的可修改性并不是意外产生的。必须有意在新代码中创建，或者小心地在重构遗留代码过程中通过遵循优秀的开发原则和实践引入。

这对于任何专业来说都是一样的。医生不可能挥挥手就神奇地治愈了病人。尽管有时候患者会自愈，但软件无法自我编写。你必须让计算机执行你的命令。

代码对可修改性的支持意味着找到合理的抽象且代码封装良好。归根结底，可修改性来自于理解所建模事物并将这些理解连带着各种特性都灌输到模型中去，让模型准确、一致。

这些实践不会替我们做设计。TDD对设计有帮助，但我们不能停止思考让TDD替我们编码。TDD是一个工具，可以帮助我们理解构建易修改代码的流程，这个流程尤为重要。

科学与艺术的一个区别是，科学常常有一定的流程（一个进程或者一个程序）。这就像根据菜谱做饭，我们可以从同样的菜谱开始，做一些不同的修改，然后做出风味各异的同一道菜肴。我们始终遵循同样的实践：如何煎炸，如何切菜然后烹饪口感更好，如何避免做出半生半熟的鸡肉，等等。

流程对软件开发来说至关重要。虽然我们面对的每个问题都各不相同，需要不同的方法来解决，但是解决软件问题的基本通用流程还是有的，有些甚至是反直觉的，就如同测试先行和最后进行设计。

13.8　第二次做好

TDD用真实的用例来定义行为。用真实的用例来驱动开发比抽象的思考要容易。它有助于构建更稳定的接口，当我们有两三个用例之后，将代码通用化会比只有一个用例来得简单得多。我喜欢这样的说法：

第二次做好。

在听到我这样说之后，开发者有时候会用奇怪的眼神看我，好像我疯了一样。我们一直被教导着凡事要**一**次做好。

但是，当我们一次做好（或者说**认为**自己一次做好）的时候，我们给自己添加了很多额外的工作。只有一个用例很难进行通用化设计。在只有一个用例的时候进行具体化的编写，在有了两三个用例之后进行通用化就相对容易了。我们可以观察各个用例之间的异同，然后进行总结，找到合理的抽象。

在天文导航中，三角定位法是一个非常常用的手段。通过多个水平线上的点或者多颗星星得到的定位，比只通过一个点得到的定位要准确得多。编写代码也是同样的道理。

如果有一个非常复杂的算法，不确定如何才能完美解决，先创建几个用例。通常在两个（最多三个）用例的辅助下，就能推导出真正的算法了。这比只通过一个用例来猜想要容易得多。

有了两个用例，就可以开始对每一步进行总结，得出正确的抽象。所以，如果只有一个用例，就直接把它编写出来，当然，是使用测试先行开发。这让我们可以用真实的单元测试来驱动行为的开发，而且在得到第二个用例要重构代码的时候也提供了安全保障。

软件是**软**的，利用这个特性我们可以更轻松地构建出更优质、更灵活的代码。这是重中之重。

相反，试图一蹴而就会有很大的压力。对于所有人来说都一样。知道可以回过头去修改、随时清理（可以在任何时候重构），会让我们很自由。

研究重构是学习如何从一种设计转变到另一种设计的最佳方式。这种转变大都很容易，理解了如何转变设计，也就意味着可以不用试图在一开始就找到最佳设计，我们可以随着重构来不断改进设计。

对于习惯了确定性的我们，这样的策略乍看之下有些奇怪。它认定我们是处在一个万事万物都在变化的难以预计的世界之中。在习惯了这种开发方式之后，可以更快速地构建更优质的代码，而不需要提前将所有的需求都准备完毕才开始构建。这让人很自由，也是因为如此，我发现它能让开发者产生共鸣。

13.9　让我们付诸实践

以下是把这些想法付诸实践的方式。

13.9.1　助你正确重构代码的 7 个策略

重构给予开发者改进设计的机会，也让管理层以廉价且低风险的方式在已有系统中添加功能。以下是帮助你正确重构代码的7个策略。

13

从已有系统中学习

重构是一种学习代码的方式，也是将学到的东西融入代码的方式。比如，将命名不合理的方法用更合理的名字取代或包裹起来，可以提高代码的可读性。同时，我们也学习到了系统是如何工作的，并且将这些理解通过提供更合理的命名融入到了新代码中。

循序渐进

低风险重构是重构方法中的子集，可以相对简单地执行。大多数可以通过开发工具进行自动化，比如在一个工程内重命名、提取、移动方法和类。我在编码的时候经常使用这些重构方法，让代码时刻反映我对所构建系统的最新理解。

在遗留代码中添加测试

所有的重构都会降低四件事情的开销：日后理解代码、添加测试、添加新功能、更多的重构。在重构的过程中会发现改进设计、添加单元测试的机会。添加更高质量的测试之后，会更有信心进行更激进的重构，进而给编写更多高质量测试创造机会，如此往复。

始终进行重构

重构是需要自始至终进行的。编程通常都是一个发现过程。也许在探索的过程中并不知道最好的方法，所以在有了更好的理解之后才有更多的机会去改进代码、更新命名等。这是保持代码容易使用的关键。如果实践TDD，就会知道在TDD过程中，重构是在编写出可用实现之后立刻进行的。这样可以提高代码的健壮性，减少维护成本，提高可扩展性。

有更好的理解后对一个实现进行重新设计

即使持续重构，也依然会在开发过程中产生技术债。当得到会影响设计的新信息或者需要实现一个当前设计不支持的功能时，可能是时候做一次大规模的重构了。也许需要大范围的重新设计和实现，让以后更容易添加新功能。

继续其他工作前进行清理

一旦完成了某些工作，应在继续其他工作前重构现有代码，让其更加健壮。在知道了每个方法的职责之后，保证它们有合理的名称来表达其意图。保证代码容易阅读、分布合理。将大方法拆分为小方法，必要的时候提取出额外的类。

重构以避免误入歧途

现如今多数生产环境中的软件都积累了大量的技术债，迫切需要重构。这看上去像是一个可怕的任务，也确实可能会如此，但是重构代码也可以非常有趣。我发现自己在重构代码时收获颇丰，在花费大量时间清理他人（或自己）的错误后，在编写新代码的时候能避免类似的错误。随着重构越来越多，我也随之成为更优秀的开发者。

重构是为了降低风险减少浪费。高效的开发团队可能花费一半的时间在重构代码上，同时也优化了他们的设计，提高了系统的健壮性，时间花得很值。因为代码被阅读的次数是编写次数的十倍以上，利用重构来清理代码会很快得到收益。

13.9.2 决定何时进行重构的 7 个策略

鉴于整个行业中需要重构的代码远远多于我们的承受能力，我们需要决定对哪些代码进行重构。如果生产环境上的软件正常工作不需要扩展，则无需重构代码。重构代码有风险和成本，所以我们希望最后收益能够抵得上开销。以下是决定何时进行重构的7个策略。

当关键代码维护不善的时候

多数软件的状况无法进行安全的重构。如果代码处在生产环境，对它进行修改，即使是看上去很小的改动，也会造成未知的破坏。因此，别去碰触遗留代码总是明智的。但是当关键代码难以理解变成累赘时，就是时候进行清理了。这种场景下，添加测试来支持更复杂的重构非常有效。

当唯一理解代码的人没空的时候

我们编写的软件应该可以让团队的其他成员容易理解和维护，但是，有时现有的代码只有那些特定的“专家”才能维护。这对公司来说不是一件好事。如果代码需要维护更新，让关键人员在继续其他工作前花时间清理代码，这可以避免后期花费更大的成本进行清理。

当有信息可以揭示更好的设计的时候

需求，以及我们对于需求的理解，都是一直在变化的。当有了更好的设计方案，而且收益比成本要高的时候，重构就是个好主意。这是一个持续进行的过程，用来保证软件整洁且与时俱进。通过一系列的重构来改进设计，是非常有效的保持软件可维护的方法。

当修复bug的时候

有些bug仅仅是拼写错误而已，而有些则代表了设计上的缺陷。很多时候，代码的bug体现了开发流程上的缺陷，或者至少是系统中一个缺失的测试。也许是因为难以编写测试所以缺失，这样的话，我们可以重构代码，让编写测试变得容易，然后补全测试。接着再修复bug，测试通过之后则一切正常。

当需要添加新功能的时候

向不兼容新功能的系统中添加新功能的最廉价、最安全的方式就是，先重构代码让系统可以兼容新功能，重构完毕之后再添加新功能。我们不会希望自己同一时间对多处代码进行修改。为添加新功能而重构代码，通常需要添加新的抽象和接口，让新功能更容易插入到现有系统中。重构之后，向代码中添加新功能就应该轻而易举了。

当需要为遗留代码写文档的时候

有些代码很难理解，在编写文档前通过简单的重构和清理就能有很大帮助。编写文档的目的就是为提高系统的保障性，这也是重构的目的之一。

当重构比重写容易的时候

将生产环境上的系统直接抛弃彻底重写，几乎从来都不是个好主意。重写一个应用通常都会比原来的系统积累更多技术债。如果重写得不够彻底，很可能也会犯之前一样的错误。重构则是一个安全的逐步清理代码的系统性方式，同时也可以保持系统持续运作。

重构有开销，而且有许多代码需要重构。为了在有限的资源下做出最好的选择，必须有针对性地重构。当需要改动代码（比如修复bug或者添加功能）的时候，通常都是重构的好时机。把握这些重构的机会，就会让代码更容易维护和使用。

13.10　总结

在代码需要修改的时候**重构遗留代码**。使用重构技巧有条理地进行修改。我们的思想应该由外部的质量监控转移到通过重构来提升代码可维护性并降低软件所有者的开销上。

本章中心思想如下。

- 学习如何有效地清理代码，以偿还技术债。
- 将为新功能创建容纳空间和开发新功能分开，可以大大简化任务，降低引入bug的风险。
- 更有效地清理代码，理解为什么在构建软件时要持续改进设计。
- 熟悉重构之后，自然会编写出更整洁的代码。

重构，以及如何正确重构，是清理遗留代码的重要手段。重构是修改或更新遗留代码的第一步，同时，也要对新编写的代码进行重构，防止它演变成遗留代码。重构也是学习现有代码库的有效方法。

第 14 章

从遗留代码中学习

我们讨论了如何编写新代码，以及优质的可维护代码是什么样的。来自面向对象编程和极限编程的务实原则和实践，会帮助开发者编写出更容易维护和扩展的代码。随着越来越多的软件开发者开始采用这些实践，软件维护的成本也会随之降低，这会让用户在软件的整个生命周期中获取更多的价值。这是个美好的设想，但是**现状**又如何呢？

现存的那些遗留代码该如何处置？那些根本没有按照可维护开发实践构建的软件又如何？

虽然我们面临的是一个严重且紧迫的问题，但本书依然关注于如何编写新代码，因为我希望让大家都理解优质的代码是什么样的——我们应该关注哪些方面让软件更容易维护。在理解什么是优质代码之前，我们没法把遗留代码重构成优质代码。

我们也讨论了我们是如何陷入这般境地的：坐在堆积如山让开发者望而生畏的遗留代码之上。

不幸的是，遗留代码危机在有所好转之前可能还会每况愈下。正如我们在第2章中看到的，低效的软件开发流程仅在美国每年就造成至少数百亿美元的损失。人们会因为糟糕的软件而丧命，而不幸的是，短期内情况不会有所改善。

当然，没办法简单地把所有的软件都用我们讨论的实践或其他实践重新加固一遍。覆水难收。但是，这9个实践会帮助我们向更专业的方向前进，从现在开始编写出更优质的软件。

如果每个人都开始使用这9个实践编写代码，让所有的新软件包都更容易维护会怎样？到什么时候那些陈旧的遗留代码会完全从系统中淘汰掉？

软件产业是一艘巨轮，让这艘巨轮调转船头需要时间。在短期内，大部分公司构建软件的方式并不会改变。Scrum对实践极限编程很有帮助。至少Ken Schwaber和Mike Beedle在《Scrum敏捷软件开发》[SB01]的封面上这样声称。但是，敏捷和Scrum被更多地当成了管理实践，其技术实践的意义却被忽视了。

有的公司会实践一些敏捷技术，**掌握**了这些实践后开发出了优秀的软件，这是很多实践敏捷的公司未能实现的目标。

软件开发很难，软件开发者必须做对很多事情后才能产生有价值的软件。这非常困难。

面向对象编程从20世纪90年代就开始流行，但只有少数的开发者能利用面向对象的特性来提高代码的可靠性。我审查过上百万行的客户代码，多数由优秀的面向对象语言Java或C#编写，但大部分的代码——而且是驱动全球关键性产业的企业级系统——是高度过程化、冗余且低质量的，所以难以维护。

我们可以引入一些实践，但这些实践变为主流还需要时间，软件开发者和管理者想要正确运用这些实践，需要理解其背后的**原则**。

敏捷和极限编程是**由**软件开发者创立的，也是**为**软件开发者创立的。从我的专业经验来看，这些技术实践和敏捷管理实践一样重要，甚至更重要，但许多组织却不怎么在意。你可以在不采用敏捷技术实践的前提下，仅仅通过敏捷管理实践获得收益，但敏捷真正的价值所在是极限编程中这些技术实践的应用。

对于向传统瀑布模型开发团队教授极限编程实践，比如测试先行和演化式设计，我有许多成功经验。他们利用这些实践取得了巨大的成功，即使他们依然在瀑布模型的框架下开发。如果一个组织的工作进度都很好，我则通常不会介绍太多的敏捷管理实践。

向现有的组织中引入敏捷的挑战之一是，许多组织有很深的组织架构，有许多和瀑布模型一样的通道和流程。将他们的信息技术部门带入敏捷实践会非常有挑战，但我发现几乎每个我合作过的团队都需要更快构建和更容易维护的软件。本书中讨论的9个实践对此非常有效。

我给微软的开发者教授了多年敏捷设计和测试先行开发，他们中的许多人都在一个伪瀑布流程中工作。无论他们是如何得到需求的，当需求到来之后，他们的开发者（至少部分开发者）会用测试先行的方式开发。这让他们的软件高度可变，可以应对在以后的开发周期中引入的新需求。

极限编程中的开发实践，诸如测试先行、重构、结对编程、设计技巧和持续集成，是软件开发成功的关键，无论采用哪种开发方法论都是如此。它们对理解问题域和精确建模提供了环境。

14.1　更好，更快，更廉价

我们看到一些数据和研究成果，它们指出软件产业内令人吃惊的失败率以及极高的软件维护成本。我们有机会研究一些新的想法、方式和实践，它们可以帮助我们摆脱这样的困境。极限编程、精益和Scrum已经出现一段时间了，已经有许多团队开始尝试用新的方式来摆脱瀑布模型。

他们的情况如何？

量化软件管理（Quantitative Software Management，QSM）协会暨卡特财团研究了采用敏捷实践的效果，发现在日程紧迫的瀑布模型项目下缺陷率会更高，有时会是四倍于平均值。高度成熟的极限编程或Scrum团队表现最佳。这些团队的缺陷率比平均值低30%～50%。[①]

① Mah, Michael. “How Agile Projects Measure Up, and What This Means to You,” Cutter Consortium, 2008. http://www.cutter.com/offers/measureup.html

在《通过测试驱动开发实现质量改进：四个工业化团队的成果和经验》①中，Nachiappan Nagappan、E. Michael Maximilien、Thirumalesh Bhat、Laurie Williams表示，他们研究的所有采用TDD的团队的缺陷率都有显著降低："IBM 团队有40%，微软团队有60% ~ 90%。"他们得出结论，TDD可以"在对开发团队的生产效率没有明显影响的前提下显著降低软件的缺陷率"。

在微软，Thirumalesh Bhat和Nachiappan Nagappan研究了TDD实践对Windows和MSN部门的影响，发表了论文《测试驱动效果评估：工业案例研究》②。他们发现，在同一组织内，采用TDD的项目比不采用TDD的相似项目，其代码质量有显著的提高（超过两倍）。

北卡罗来纳州立大学的Boby George和Laurie Williams研究了两组软件开发者，让他们进行"小型 Java 程序"开发，从中发现TDD开发者会产出更高质量的代码，"功能性黑盒测试的通过率高出18%"。虽然TDD组花费了16%的额外开发时间，但TDD开发者的测试用例达到了"98%方法、92%语句、97%分支"的覆盖率。在和开发者交流的时候，发现92%的开发者认为"TDD 会产生更高质量的代码"。③

有人会告诉你实践TDD能减少缺陷，但是有成本。你会编写比产品代码两倍还多的测试代码，所以自然而然人们会认为这降低了开发速度，但这是个错误的假设。这种想法认为影响软件开发速度的瓶颈因素是打字。

但这不是真的。询问任何一个开发者，考察任何一个项目。开发者花费的大部分时间**不是**在编码上，而是在以下方面：阅读需求文档、编写文档、开会，还有最耗费时间的排查bug。

测试先行开发并不需要上述的大部分工作，而是让开发者更多地进行他们最喜欢的事情：编写代码。测试就是代码，但不是一般的代码。测试代码很重要，是因为它们同时也是要实现功能的需求定义。

提高效率并不意味牺牲质量。相反，它们通常都是共同进退的。在《在真实环境中探索极限编程：工业级案例研究》④中，Lucas Layman、Laurie Williams、Lynn Cunningham发现，比较同样一个产品的两次发布，有"50%的效率提升，65%的发布前质量提升，35%的发布后质量提升"。一次发布是在团队采用极限编程方法论之前，一次发布是在采用极限编程两年之后。

---

① Nagappan, Nachiappan, Maximilien, E. Michael, Bhat, Thirumalesh, and Williams, Laurie. *Realizing Quality Improvement Through Test-driven Development: Results and Experiences of Four Industrial Teams*. Springer Science & Business Media, 2008. http://link.springer.com/article/10.1007%2Fs10664-008-9062-z

② Bhat, Thirumalesh, and Nagappan, Nachiappan. "Evaluating the Efficacy of Test-Driven Development: Industrial Case Studies." http://dl.acm.org/citation.cfm?id=1159787

③ George, Boby, and Williams, Laurie. "An Initial Investigation of Test Driven Development in Industry." Department of Computer Science, North Carolina State University. http://staff.unak.is/andy/MScTestingMaintenance/Homeworks/STMHeima7TestDrivenDevelopment.pdf

④ Layman, Lucas, Williams, Laurie, and Cunningham, Lynn. "Exploring Extreme Programming in Context: An Industrial Case Study." http://dl.acm.org/citation.cfm?id=1025140

量化软件管理协会暨卡特财团研究了福莱特公司的极限编程实践，发现“时间周期比行业规范戏剧性地减少了五个月，质量提高了两倍（缺陷降低了一半）”。不仅如此，福莱特公司还节省了一百三十万美元。将这个数值乘以六次发布，整整节省了七百八十万美元。

关于TDD对代码质量影响的数据比较缺乏。在我找到的一篇论文《对开源项目进行面向对象质量度量，以量化评估测试驱动开发》[①]中，Ron Hilton研究了大量使用和未使用TDD的开源项目的代码质量。他发现使用测试驱动的项目，内聚性指标高出21.33%，耦合度指标高出10.05%[②]，复杂度降低了30.98%。

有着如此高的潜在提升空间，人们会觉得多数公司都会采用极限编程实践，但这些数据并未被广泛认知。《2013年Scrum状态报告》显示，受访的公司中40%使用Scrum，15%使用看板，11%使用精益，只有7%使用极限编程。这些实践有一定的学习曲线，需要花费精力来掌握，但潜在的收益是真实且巨大的。

14.2　不在不需要的事情上花钱

我们都是凡人，总会犯些小错。但是在严格按照指令执行的计算机中，一个小错误可能引发大问题。计算机不知你的真正意图是什么。它们不是传译或者翻译，又或者仅仅将代码当作建议或指导，而是盲目地执行特定的指令。所以，如果遵循特定的规范保证程序正确执行，然后持续进行测试，我们就可以（通常是非常快速地）修复任何bug然后继续工作。耽误不了多少时间。

但是许多软件开发团队，无论敏捷与否，并不实践TDD，而是依赖于独立的质量保证阶段，质量保证阶段可能需要进行两周，开发者才能得到关于他们代码的反馈，同样的小bug可能会是致命的。在修复bug之前，开发者可能要花费大半天的时间重新熟悉几周前开发的代码。如果时间就是金钱的话——我们雇用开发者、质量保证工程师和其他相关人员的时候，时间就是金钱——花费几秒修复bug会比花费一两天要廉价得多。

如果我们通过优秀的开发实践，而不是通过独立的质量保证阶段来**保证**质量，就可以做出戏剧性的转变。我们可以更好地关注于健壮性。这是降低软件所有者成本的唯一出路。**整洁**、可测试代码的维护和扩展成本更低。这是底线。

如果本书讨论的内容可以消除在前期需求阶段（三分之一到一半的开发时间）的花销，并解决令人生畏的集成问题和整个调试阶段（另外三分之一的开发时间），那么我们就没有因为编写测试而耗费更多时间。我们可以**省下一半以上的时间和精力**。

如果是这样的话，你是否愿意将省下的时间投入到重构代码上，清理代码让其质量更好？如

① Hilton, Ron.“Quantitatively Evaluating Test-Driven Development by Applying Object-Oriented Quality Metrics to Open Source Projects.” http://www.nomachetejuggling.com/2009/12/13/quantitatively-evaluating-test-driven-development/

② 并不是说高耦合，论文中的度量方式是数值越高耦合度越低。——译者注

果你愿意这样做，就会领先于传统的开发模式，开发出更低成本、更容易维护的产品。

我并不是要你完全摒弃质量保证。因项目而异，可能需要大量的质量保证工作，可能完全不需要。但无论怎样，我们都应该关注于通过优秀的技术实践来提高质量，力求将验证待发布版本的工作自动化。这样做会大大降低开发和维护代码的成本。

14.3 循规蹈矩

软件开发中，有一条一切都清晰明了的循规蹈矩之路。我们应该力争走上这条路，这样，当我们发觉不对劲时，立刻就知道自己误入歧途了。

我们承受不起一丁点的误入歧途。丝毫的偏差可以导致我们前功尽弃。

软件开发者最大的疏忽之一就是会在设计中丢失实体。在建模中缺失实体会让我们不知道该将行为置于何处，所以模型变得扭曲了。设计和理解会有偏差，而理解又和要建模的事物有偏差，这就是为什么会有那么多的bug、糟糕的代码等问题。这也就是为什么我会强调最开始的理解，应该**主动**而不是**被动**。四处忙着填坑救火的场景，我们在工作中都时常见到，这总让我觉得沮丧。

软件开发不应该是一个被动的过程。必须是一个深思熟虑的过程，但并不意味着节奏必须很缓慢。事实上，先编写测试并不会让我们慢下来。我知道这是真的，因为我见证过许多次。正确利用这个过程，理解它，度过它的学习曲线——所有新事物都会有学习曲线的。

我知道很多开发者在采用测试先行开发后，效率有了很大提升。他们开始意识到开发并不是在敲代码。在软件开发中拖慢速度的是调试。编写、阅读、理解需求文档，诸如此类，所有这些非编码活动进入了软件开发流程。合理采用TDD能解决这些挑战，用编写单元测试替代这些非编码活动。正如我们之前所说，测试也是代码，所以TDD意味着开发者花更多时间在编码上，进而产生更大的价值。

但是依然面临着挑战。

很难学习如何正确使用这些实践。这方面的优秀书籍并不多，也有一定的学习曲线。我见过有的团队挣扎了十二个月甚至更多，试图学习这些实践，或者没办法有效实行，那是因为他们没理解我们前面讨论过的这些特性。

理解**为什么**这些实践如此重要，而不仅仅是理解这些实践**是什么**，这会让团队更快掌握它们。一旦团队上了正轨，如果他们和另外一些熟练使用这些实践的开发者一起工作，就能很快提高速度，那么通常只需要几个月就能利用测试先行高效地工作了。

但真正的挑战来源于采用TDD之前编写的大量代码。之前花费数年挖的坑也需要数年来填。但好消息是，一旦开始填坑、偿还技术债，也就开始体会到**真正的**高效了。

14.4　提升整个软件行业

我们需要理解软件开发中的轻重缓急。我们需要分享那些可学习且容易理解的原则和实践，这样才能为软件开发建立规范——为了让它成为真正的专业性行业。

这和其他工程领域的道理一样。经验丰富的水电工都遵循着久经考验的标准和实践，而确立这些标准和实践的人们不单是以最快、最高效地完成工作为目的。他们也在考量其他因素，比如公共安全。他们必须合理接入市政供水和排水系统以及电网，而且这些标准必须是通用且可执行的。

电工对安装一个照明设备的收费肯定不如重新将整个房子走线多。所有人都明白在物理世界中，有些东西容易修改，有些则不容易。

但是，在无法触摸的虚拟世界中，多数的非开发者难以理解这一点，而且很难分辨哪些容易修改哪些不容易修改。

20世纪90年代，软件开发社区开始展现出封闭的特性。可以构建更优质软件的新技术被视为竞争优势而不乐于分享，但这种竞争方式并不是良性的。公司应该对他们的专有软件进行保密，而不是他们的开发方法。我们应该在整个行业内分享我们的开发方法。这是唯一能够快速提高整个行业的方式。对于其他行业，这个方法行之有效，对于软件行业也应如此。幸运的是，我看到有一些大公司已经免费公开了他们的方法。

世界需要我们软件行业的从业者挺身而出，行业中——软件开发中所有专业的——所有人都需要互相扶持。我们必须自我管理。如果不这样做的话，政府可能会出面管制，那将会是灾难性的。

一所医院的医生如果将能够救人一命的信息对其他医院的医生保密，那是不道德的。软件开发者也是一样。我们开发的一些产品真的会挽救生命，所以我们必须分享知识。如果我们提高了整个行业的专业性，就会得到正反馈。我们分享方法论、模式、原则、实践。我们并不需要分享商业机密和专有信息。

软件一直处在未知世界之中，技术、原则和理论在不断进化，这种状态将会一直持续。软件开发是一个年轻的行业，即便我们发展迅速，也依旧前路漫漫。

对此没有简单的答案，但可以用我们的才智来解决这些问题。让我们开始公开讨论和共享标准，敞开心扉珍视那些重要的事情。

最终，构建一个健康的行业，就像构建一个健康的社会一样，需要每个人都参与其中。任何组织都是依靠其成员运作的，我们已经见到以前所未有的方式进行软件开发的新型组织正在崭露头角。开源、知识共享协议，以及诸如GitHub[①] 这样的工具，给各种工具和库提供了免费获取渠道。我们有了改变整个行业的基础。剩下的只是愿不愿意使用的问题。而且这种转变正在发生着。

① GitHub. https://github.com

软件开发者如何分享技巧？如何高效地从他人那里学习？开发者需要更多地分享思想和互相学习。我们需要在学校开设更多课程，也需要给校园外的从业者提供学习场所。我们必须更加重视开发技巧，通过提高待遇来让软件开发更能吸引且留住优秀人才。

在过去的几十年中，我们，所有软件行业的从业者，有了长足的进步。我们创造出了更容易维护的代码，不仅仅是通过极限编程，也通过一些没有那么极限的技术和实践，比如设计模式、软件工艺化运动、整洁代码运动以及其他实践。我们学到了什么？

这些思想是统一的，这十分令人振奋。设计智慧是不断精进的，其展现出来的一致性是个好征兆。本书讨论的所有原则和实践都是高度统一的，它们都有助于构建更容易测试、维护和扩展的软件。这说明我们已经开始缕清思路，开始理解如何更有效地构建软件了。

软件是独一无二的，我们必须承认这一点。我们在学习高效构建软件的方式，自动化一些繁重重复的任务，提高整体的开发效率，以便我们可以专注于工作中真正具有创造性意义的部分。前途一片光明，但我们尚未走出迷雾。

在以后，我们也许会为发生的灾难而怪罪软件，但是这就和怪罪泰坦尼克号上的铆钉或者太空舱上的瓷砖一样。损坏总会发生，而且常常发生在薄弱环节，但这并不是事情的全部。TDD、结对和其他技术实践可以有所帮助，但软件开发没有放之四海而皆准的方法。在很多场景下，面向对象和测试驱动开发就不适用。

它们归根结底不过是工具。工具不能决定一个专业。它们仅仅是一方面而已。

世界上的许多问题都是过程化的。许多事物是有层级的，但是有些不是。我们需要找到建模非层级非过程化事物的方式，现在已经有许多范式可以使用。我们在本书中仅着重于面向对象的范式，但是还有许多其他的软件开发范式，在有了大规模并行计算以后，出现了更多的选择。不过范式也只是工具而已。敏捷也是工具。所以我想抛开敏捷和Scrum这样的词语说一句：

我们是软件开发者，利用现有工具尽我们所能开发最好的软件。

14.5 超越敏捷

如果有比敏捷更优秀的方法论出现，我将第一个投入其中。即便我是一个“敏捷主义者”，并不意味着抛弃了以前所有的实践。我依然从年轻时习得的技能中受益。

若干年后，本书中讨论的实践才会变成主流。测试先行开发的巨大价值会使它成为多数软件开发遵循的实践。也许和我们现在的TDD不太一样，也许使用不同的工具或语言，但是在编写代码时进行独立验证会纳入到软件开发过程之中，甚至纳入到开发语言和IDE中。

进行TDD的优势很大，所以这样做是有意义的。我们今天大部分开发者的编程方式在未来看来，就好像今天我们回顾过去用机器码编程一样。当然，你可以这么做，但为什么要这么做？通过二进制开关输入操作码是艰难而又乏味的过程。这就是为什么我们需要编译器，同样，单元测

试也是用来捕获那些人类无法避免的错误。但是，归根结底，只有人类才能创造软件。

软件开发和一般的体力劳动有天壤之别，它需要一系列技巧和创造力才能成功。如果想要提升软件质量，就必须提高我们的标准以及那些达到标准的开发者的待遇。如果我们这样做，人们就会应时而起。我们可以随心所欲，这在软件开发中比其他行业更真实。软件是纯粹的思想产物。它源自我们的大脑，通过我们的手指，输入到计算机中。

而它掌控着一切。

无论你是哪个行业的，银行、保险、运输还是金融，你都离不开软件。软件掌控着一切行业，你的行业也不例外，行业中开发出的商业软件往往成为竞争中的“秘密武器”。

软件是一切事物的中心，是我们社会中的至上之力（deus ex machina）。它可能推动我们，也可能阻碍我们。所以，优化软件让它更有价值，对我们所有人来说都有好处。然而，我们今天创建的软件正迅速地消耗着我们的资源。我们过去未关注建模的精细准确，这让我们今天深陷其中。我们创建的模型节省空间或者执行速度快，但由于没有学习过如何准确进行领域建模，软件虽然按照期望执行但是难以调整，而事实表明，我们需要在软件的生命周期内频繁调整。

如果我们偷懒没有准确建模，我们的理解就会有误。我们会失去大局观，当我们忘记了事物的缘由的时候，事情就会变得僵化。

14.6　将理解具象化

没有编写优秀软件的现成配方，以后也不会有——正如没有关于著书、写歌、编剧或者画画的现成配方。只有一些可以遵循的指导方针，可以采纳的技术，可以学习（然后打破）的规则，以及可以使用的实践。

这些都是工具，和做其他事情一样，结果取决于如何发挥自己的技艺，以及如何运用手里的工具。工具越强大，也就越容易误用。用链锯伐木比手锯更快，同样也更容易伤到自己。这个物理世界的隐喻在虚拟世界同样适用。工具越强大，也就越容易误用，所以我们必须小心使用手中的工具，才能保证它正确发挥作用。

我们处在许多物理科学突破的边缘。现有的量子计算机带来无数的处理器和新技术，即将让这些突破黯然失色。物理科学正在无与伦比地迅猛发展，而虚拟科学——软件开发——似乎停滞不前了。我们现在构建软件的方式大体上和半个世纪之前没什么两样。汇编器、编译器，甚至面向对象的语言，都没有从根本上改变设计和构建软件的方式。

测试驱动开发代表了构建软件方式的另一种改变，更多的改变也会到来。如果想要充分利用那些新的物理计算机，就需要更多的改变。如果软件想要跟上硬件，我们必须愿意进行指数级的成长，也就意味着，和现在相比，我们要大规模地减少错误并提高可靠性。

软件行业中最令人激动的尖端科学尚未到来。我们今天编写的软件以将来的标准看会被称为原始。解决像天气建模或提供真正的智能系统这样的难题，远远超越了当前的软件技术。

当我二十出头刚刚入行的时候，我们曾经以为会思考的计算机会在几年内出现。今天，我们意识到我们没有处在软件开发的黄金时期，反而更像是在软件开发的“石器时代”。但我们正在飞速成长，经历了小型的“文艺复兴”。真正实现人工智能的设想也许需要五十年，也许需要五百年。

我们这个行业下一步需要的是联合起来构建统一的基础。在我们探索星空之前，必须先脚踏实地。软件必须要比现在更可靠，为此，必须更重视软件的可靠性。为了能在新型计算机上编程或解决难题，必须想方设法交付出和今天大不相同的软件。我们不能急于求成。相反，必须回过头追求更高效的**理解**。这是优秀软件的特性。

优秀软件是具象化的理解。

14.7 成长的勇气

我们倾向于重复做一件事，即使结果不尽如人意。我们这样做不是因为疯了，而是因为对未知领域充满恐惧。我们有足够的理由害怕未知。混乱是难以应对的，所以我们宁愿坚持着已知的东西，即使已知的并不理想。

但只有在未知当中才能发现新的事物——前人没有发现的事物。探索未知需要勇气。也许勇气是伟大思想者最伟大的特质之一，也许听上去奇怪但确实如此。勇气的种类有很多。就像《激情与偏见》[Ros78]的作者Leo Rosten曾经说过的：“不知恐惧的人并非真正的勇敢，因为勇气是直面想象之物的能力。”我们的想象力越强，也就需要越多的勇气。

勇气也不仅仅是愿意面对恐惧。我们需要勇气来面对成功或者失败。任何会威胁现状的事物都会引发潜意识的恐惧。为了克服恐惧，仅仅自我说服是不够的。我们必须要**感到**安全。言语上的鼓励有所帮助，但是，针对潜意识的恐惧，我们需要用潜意识的语言来解决——用想象和隐喻。

如果回想起之前类似的情形并且幸存了下来，意识就会停止恐惧。即使是最坏的情况，当我真的害怕无法自控的时候，我就硬着头皮前进。恐惧是种自我保护机制。重视恐惧带来的消息，可以防止恐惧让我们无法动弹。

我希望我们能有勇气做出正确选择，我们对它的重要性的理解事关重大。坦率地讲，我可以在几个小时内教授一位资深开发者这9种实践。我们可以在十页纸内把它们写下来。但是仅仅知道做**什么**还不够。我们需要知道**何时**做、**如何**做，而最重要的是，**为什么**这么做。

我们必须理解软件的本质。一旦理解了，让其显而易见了，就能形成和其他事物一样的基本认知，也就没那么可怕了。

管理层无法把勇气强加给一个软件开发团队。勇气必须来自团队内部：每个开发者都意识到这些实践不是一个负担，而是可以帮助他们构建更好的软件。我所知道的多数成功实行这些实践的团队都是自己做的决定，他们说："经理，我们要这么做。"而不是被迫去做。

如果我们理解了这些实践背后的原则，它们就不是负担，完全不会拖慢开发速度。我认识的顶尖开发者编写的代码是高度可维护、**整洁**的代码，而且他们编写代码的速度比其他人都要快。

我们希望能以一种方式编写出引以为傲的软件，就像其他创造性成果一样。我希望能够打破"快速即是廉价"的观念。如果不注意这些优秀实践，也许短期看上去快一些，但是纸牌屋的倒塌会比预期的快很多。只需要看看本书第一部分的内容，就能知道我们有多少失败的记录了。软件行业的失败率要比成功率高很多，但并非注定如此。我们的客户越来越精明，他们不会想成为内部测试者，也不会为"优先"测试我们的软件而付费。

我想说的是，让我们后退一步，想一想我们究竟在做什么。构建软件是一项复杂的工作，但是，越多地思考我们在做什么，是如何做的，为什么要这么做，就越能做好我们的工作。首先就是要有勇气认清当前的缺陷，尝试新的实践。

我真诚希望本书中讨论的内容能够帮助你构建更优质的软件。但我也同样希望有一天这些想法会被更优秀的想法超越。这是变革的本质，也是我们追求的。这些实践比我们现有的实践有所突破。但这仅仅是一个开始而已。

我见过许多大公司构建软件的方式。我也和许多500强企业以及其他企业合作过。我知道IBM、微软、雅虎是如何构建软件的，因为我和来自它们及其他公司的上千个软件开发者共事过，教授他们本书中讨论的实践。我见证过这些实践在真实环境下取得成功。

同样我也见过太多的组织一心要重新发明轮子。软件开发中只有很少的标准和通用实践。软件开发者熟知一门通用的编程语言，但通常没有通用的设计方法，或者通用的审美标准，甚至对他们构建的功能都没有通用的目标。我们对"优秀软件"的定义各有不同。

就好像盲人摸象一般。我试图把从客户那里学到的技术分享给其他人。这是作为一个咨询师最大的好处之一：我可以从我的客户那里学习。

我学到的是，解决问题的方式有许多种。人们思考问题和解决问题的方式各不相同，其中的差异让人惊奇。哪个才是正确的？

全部都是！

每个方式都有其价值，对不同的方式进行研究，可以获得适用于其他问题的工具和技巧。

在软件当中，没有人能告诉我们什么是正确的。我们必须自己搞清楚。用新的方式工作可能会令人激动，但需要不断开拓新道路也会令人沮丧。不是每个人都能成为一个开发者或探索者。感谢上天，我们有Magellan、Lewis和Clark这样的探险家，以及Kent Beck、Ward Cunningham这样的软件先驱。

有些人真正理解如何构建软件。他们构建的软件成本低廉且成功率高。这些成功事例告诉我们，有更优秀的构建软件的方法。我们之前所学的实际上是传统的质量保证模型。它充满了经验主义的气息，同时给了我们线索，让我们对自身和周围的世界有着更深刻的认识。

工业革命带来了大规模的制造业，统一而持续。我们正在步入信息革命，它需要一个崭新的方式。革命就在我们身边，要求我们从根本上改变原来的思考方式，注重那些和上个世纪的实利主义思想几乎截然相反的价值观。信息革命要求的不是统一和持续，而是个体和创新。

参考文献

[Ad 11] Gojko Adžić. *Specification by Example*. Manning Publications Co., Greenwich, CT, 2011.

[Bai08] Scott Bain. *Emergent Design: The Evolutionary Nature of Professional Software Development*. Addison-Wesley, Reading, MA, 2008.

[BE12] Daniel Brolund and Ola Ellnestam. *Behead Your Legacy Beast: Refactor and Restructure Relentlessly with the Mikado Method*. Daniel Brolund, Ola Ellnestam, http://www.agical.com, 2012.

[Bec00] Kent Beck. *Extreme Programming Explained: Embrace Change*. Addison-Wesley Longman, Reading, MA, 2000.

[Bec02] Kent Beck. *Test Driven Development: By Example*. Addison-Wesley, Reading, MA, 2002.

[Bro95] Frederick P. Brooks Jr. *The Mythical Man-Month: Essays on Software Engineering*. Addison-Wesley, Reading, MA, Anniversary, 1995.

[Coh04] Mike Cohn. *User Stories Applied: For Agile Software Development*. Addison-Wesley Professional, Boston, MA, 2004.

[FBBO99] Martin Fowler, Kent Beck, John Brant, William Opdyke, and Don Roberts. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, Reading, MA, 1999.

[Fea04] Michael Feathers. *Working Effectively with Legacy Code*. Prentice Hall, Englewood Cliffs, NJ, 2004.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, MA, 1995.

[Gla08] Malcolm Gladwell. *Outliers: The Story of Success*. Little, Brown and Company, New York, NY, USA, 2008.

[Jon95] Capers Jones. *Patterns of Software System Failure and Success*. Intl Thomson Computer Pr (Sd), London, UK, 1995.

[Mey97] Bertrand Meyer. *Object-Oriented Software Construction*. Prentice Hall, Englewood Cliffs, NJ, Second edition, 1997.

[Pug05] Ken Pugh. *Prefactoring*. O'Reilly & Associates, Inc., Sebastopol, CA, 2005.

[Pug11] Ken Pugh. *Lean-Agile Acceptance Test-Driven Development: Better Software Through Collaboration*. Addison-Wesley, Reading, MA, 2011.

[Ros78] Leo Rosten. *Passions and Prejudices*. McGraw-Hill, Emeryville, CA, 1978.

[SB01] Ken Schwaber and Mike Beedle. *Agile Software Development with Scrum*. Prentice Hall, Englewood Cliffs, NJ, 2001.

[SW07] James Shore and Shane Warden. *The Art of Agile Development*. O'Reilly & Associates, Inc., Sebastopol, CA, 2007.

[WK02] Laurie Williams and Robert Kessler. *Pair Programming Illuminated*. Addison-Wesley, Reading, MA, 2002.

本书从全新的视角展现了现代软件开发流程。工程师们会在其中找到解决日常问题的方案，而非工程师们可以对软件开发中所面对的挑战和困难有所认识。

—— **Stas Zvinyatskovsky**
埃森哲公司资深首席软件架构师

David帮我们认清了我们是如何陷入此番境地的。他给出了行之有效的理论和工具，也提出了一些值得深刻思考的问题。对于关心软件开发的人来说，本书是一份厚礼。要善用它。

—— **Ron Jeffries**
RonJeffries.com

如果你想要优化软件交付流程，但是感觉到裹足不前、无能为力，那么这本书正适合你。对于开始频繁迭代交付以及尝试采用敏捷却未见显著效果的人来说，这是本好书。

—— **Gojko Adzic**
Neuri Consulting LLP公司合伙人

本书讨论的内容可以帮我让客户更加满意，也能让他们在需求出现变化时一直开心。

—— **David Weiser**
Moz软件工程师

对于所有的开发者和管理者来说，这都是一本好书，而且适用于各类公司的各类代码。

—— **Troy Magennis**
Focused Objective CEO

David的解释清晰明了，我甚至希望开发团队的管理者以及开发定制软件的公司领导们也能看看这本书。理解这些实践，将能构建出更经济、更易维护和扩展的软件。

—— **Jim Fiolek**
黑骑士金融服务公司软件架构师

书中的各种观点令人欣慰。如果可以让人们都遵从这些原则，我们的生活以及软件开发会变得更加轻松惬意。

—— **Nick Capito**
Unboxed Technology公司软件开发总监

我们努力让每一行代码成为真实产品的一部分。要找出让我们误入歧途的原因。要找到合适的方法，让你的团队在现在以及今后能更有效地开发出客户真正想要的产品。

—— **Michael Hunter**
极客，骇客，首席工程师，架构师

The Pragmatic Programmers

图灵社区：iTuring.cn
热线：(010)51095186转600

分类建议　计算机 / 软件开发 / 敏捷

人民邮电出版社网址：www.ptpress.com.cn

ISBN 978-7-115-46776-8

ISBN 978-7-115-46776-8
定价：55.00元

看完了

如果您对本书内容有疑问，可发邮件至 contact@turingbook.com，会有编辑或作译者协助答疑。也可访问图灵社区，参与本书讨论。

如果是有关电子书的建议或问题，请联系专用客服邮箱：
ebook@turingbook.com。

在这可以找到我们：

微博 @图灵教育：好书、活动每日播报
微博 @图灵社区：电子书和好文章的消息
微博 @图灵新知：图灵教育的科普小组
微信 图灵访谈：ituring_interview，讲述码农精彩人生
微信 图灵教育：turingbooks